



计算机类本科规划教材
教育部—微软精品课程教学成果

汇编语言简明教程

◆ 钱晓捷 编著



- 基于80x86系列处理器的个人计算机
- 基于DOS/Windows操作系统
- MASM32和Visual C++集成化开发系统



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

<http://www.phei.com.cn>

计算机类本科规划教材

汇编语言简明教程

钱晓捷 编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书基于 MS-DOS 环境和 MASM 汇编程序讲解 16 位 8086 处理器基本指令及其汇编语言程序设计, 内容包括汇编语言基础、处理器基本指令和汇编语言常用伪指令以及顺序、分支、循环、子程序结构, 在此基础上, 逐步展开 32 位指令编程、Windows 编程、与 C++ 语言的混合编程, 并介绍浮点、多媒体及 64 位指令。

本书主要面向普通高等院校的计算机以及电子、通信和自控等电类专业, 可用做“汇编语言程序设计”课程的教材或参考书。本书具有“重点明确、突出实践、深入浅出”等特色, 使其还能很好地适合远程教育、成人教育、自学考试等本科或专科(含高职)学生, 也适合计算机应用开发人员、希望深入学习汇编语言的普通读者作为入门教材。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有, 侵权必究。

图书在版编目(CIP)数据

汇编语言简明教程 / 钱晓捷编著. —北京: 电子工业出版社, 2013.5

计算机类本科规划教材

ISBN 978-7-121-20184-4

I. ①汇… II. ①钱… III. ①汇编语言—程序设计—高等学校—教材 IV. ①TP313

中国版本图书馆 CIP 数据核字(2013)第 075970 号

策划编辑: 章海涛

责任编辑: 章海涛 特约编辑: 何 雄

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1 092 1/16 印张: 18.75 字数: 480 千字

印 次: 2013 年 5 月第 1 次印刷

定 价: 39.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前 言

汇编语言使用处理器指令编程，是一种底层程序设计语言。本书使用 80x86 系列处理器的个人计算机、基于 DOS/Windows 操作系统和 MASM 汇编程序，学习汇编语言程序设计，形成如下表所示章节。表中内容提要亦是教学重点，体现了本书的逻辑主线。

目 录	内容提要（教学重点）
第 1 章 汇编语言基础	在了解个人计算机软硬件系统的基础上，熟悉通用寄存器和存储器组织，掌握汇编语言的语句格式、程序框架和开发方法
第 2 章 数据表示和寻址	在理解计算机如何表达数据的基础上，熟悉汇编语言中如何使用常量和变量，掌握处理器指令寻址数据的方式
第 3 章 通用数据处理指令	熟悉处理器数据传送、算术运算、逻辑运算和移位操作等基本指令，通过程序片段掌握指令功能和编程应用
第 4 章 程序结构	以顺序、分支和循环程序结构为主线，结合数值运算、数组处理等示例程序，掌握控制转移指令以及编写基本程序结构的方法
第 5 章 模块化程序设计	以子程序结构为主体，围绕二进制、十六进制和十进制数码转换实现键盘输入和显示输出，熟悉子程序、文件包含、宏汇编等各种多模块编程的思想
第 6 章 32 位指令及其编程	了解 80x86 系列处理器发展概况，理解 32 位处理器的运行环境，熟悉新增的 32 位特色指令，掌握 DOS 环境使用 32 位指令的特点
第 7 章 Windows 编程	熟悉汇编语言调用 API 函数的方法，掌握控制台输入输出函数。了解 MASM 的高级特性，以及 Windows 图形窗口程序的编写
第 8 章 与 Visual C++ 的混合编程	了解嵌入汇编和模块连接进行混合编程方法，理解堆栈帧的作用，熟悉汇编语言调用高级语言函数和开发调试过程
第 9 章 浮点、多媒体及 64 位指令	理解浮点数据格式、多媒体数据格式及 64 位编程环境的特点，了解浮点、多媒体和 64 位等特色指令

汇编语言能够直接有效地控制硬件，便于编写代码量小、运行速度快的高效程序，在计算机及相关专业的教学和许多应用场合中具有不可或缺的作用。“汇编语言程序设计”也一直作为专业基础的必修教学内容。然而，随着计算机技术发展和高等教育教学改革，传统的教学理念、手段和方法都需要进步。基于“汇编语言在底层但不低级”的教学理念和“汇编、汇编一定会编”的精神，作者总结十多年课程主讲和教材编写的实践经验编写本书，并形成如下特色。

1. 删繁就简、重点明确的教学内容

汇编语言的教学目的是从软件角度理解计算机硬件工作原理，为相关课程提供基础知识，同时全面认识程序设计语言，体会底层编程特点，以便更好地应用高级语言。所以，本书没有将汇编语言作为一个通用程序设计语言那样展开，没有详尽论述所有处理器指令、全部汇编伪指令，而是选择处理器通用的基本指令和反映汇编语言特色的常用伪指令；没有引出复杂的程序格式、详述程序框架的每个细节，而是侧重指令功能和编程思想、特别介绍相关硬件工作原理。这样一方面能够降低教学难度、易于学生掌握，另一方面使得教学内容更加实用、便于学生实际应用。

2. 贯穿始终、突出实践的教学过程

本书将上机实践贯穿始终，教学内容融入了约 80 个例题程序和约 70 个习题程序。第 1 章在必要的寄存器和存储器知识后，就引出汇编语言开发环境，介绍汇编语言的语句格式、源程序框架和开发方法，并编写具有显示结果的程序。第 2 章结合数据编码、引出汇编语言的常量定义和变量应用，利用汇编过程生成的列表文件，自然掌握常用伪指令和数据寻址。第 3 章分类学习处理器基本指令，特别利用 DEBUG 调试程序直观体会指令功能，同时逐渐编写特定要求的程序片段。第 4 章以程序结构为主线，编写数值运算、数组处理、字符串操作等程序，同时使用 DEBUG 调试程序进行可执行程序调试。第 5 章围绕二进制、十六进制和十进制数码转换子程序，掌握模块化编程方法。第 6 章 32 位编程、第 7 章 Windows 编程、第 8 章混合编程、第 9 章浮点指令，从不同方面强化程序设计能力。

为方便教学和初学者入门，本书构建了一个简单易用的 MASM 开发环境（详见本书第 1 章），无须安装和配置，直接复制就可使用。第 2 章到第 5 章还运用 DEBUG 调试程序，学习程序调试方法。第 7 章和第 8 章分别引出了 MASM32 和 Visual C++ 集成化开发系统，说明使用汇编语言开发 32 位 Windows 应用程序和混合编程的方法。

3. 循序渐进、深入浅出的教学原则

为了便于学生理解和掌握，也便于教师实施教学，本书以“循序渐进、难点分散、前后对照”为原则，做到“语言浅显、描述详尽、图表准确”，内容编排具有诸多特色。

例如，本书从简单的 16 位 8086 处理器入手，通过前 5 章掌握基本的，也是最核心的使用指令进行编程的方法，然后从第 6 章开始才引出相对复杂的 32 位处理器指令。高级编程技术如宏汇编、条件汇编等随着模块化编程引入，结构变量和 MASM 高级特性等也是随着 Windows 编程、结合应用展开，直到混合编程、浮点和多媒体指令等相对较为艰深的教学内容。

再如，将处理器指令和汇编伪指令分散于各章教学内容之中，引出列表文件暂时避开调试程序，配合屏幕截图详述 DEBUG 调试程序的操作过程。完整的程序尽可能具有交互性和趣味性，适当对比高级语言，并揭示底层工作原理、理解问题的来龙去脉。每章都编制丰富的习题，满足课外练习、上机任务和试题组织。

本书由郑州大学信息工程学院钱晓捷老师主编，同时衷心感谢关国利、张青、张行进、穆玲玲等老师多年来的合作和帮助，热诚期待广大师生和读者的反馈。

“大学微机技术系列课程教学辅助网站”(<http://www2.zzu.edu.cn/qwfw>) 提供本书的教学课件（电子教案）、配套软件（含示例源程序文件）等辅助资源。

作 者

目 录

第 1 章 汇编语言基础	1
1.1 个人计算机系统概述	1
1.1.1 计算机的硬件	1
1.1.2 计算机的软件	3
1.1.3 程序设计语言	4
1.2 8086 处理器	7
1.2.1 8086 的功能结构	7
1.2.2 8086 的寄存器	8
1.2.3 8086 的存储器组织	11
1.3 汇编语言程序的格式	15
1.3.1 指令代码格式	15
1.3.2 语句格式	16
1.3.3 源程序框架	18
1.4 汇编语言程序的开发	23
1.4.1 开发环境	23
1.4.2 开发过程	27
1.4.3 列表文件	29
习题 1	31
第 2 章 数据表示和寻址	34
2.1 数据表示	34
2.1.1 数制	34
2.1.2 数值的编码	37
2.1.3 字符的编码	38
2.2 常量表达	40
2.3 变量应用	43
2.3.1 变量定义	43
2.3.2 变量属性	48
2.4 数据寻址方式	51
2.4.1 立即数寻址	51
2.4.2 寄存器寻址	54
2.4.3 存储器寻址	55
2.4.4 数据寻址的组合	61
习题 2	62

第 3 章 通用数据处理指令	65
3.1 数据传送类指令	65
3.1.1 通用传送指令	65
3.1.2 堆栈操作指令	67
3.1.3 其他传送指令	70
3.2 算术运算类指令	75
3.2.1 状态标志	75
3.2.2 加法指令	77
3.2.3 减法指令	79
3.2.4 乘法和除法指令	80
3.2.5 其他运算指令	83
3.3 位操作类指令	85
3.3.1 逻辑运算指令	85
3.3.2 移位指令	88
习题 3	91
第 4 章 程序结构	97
4.1 顺序程序结构	97
4.2 分支程序结构	103
4.2.1 无条件转移指令	103
4.2.2 条件转移指令	106
4.2.3 单分支结构	111
4.2.4 双分支结构	112
4.2.5 多分支结构	114
4.3 循环程序结构	117
4.3.1 循环指令	117
4.3.2 计数控制循环	119
4.3.3 条件控制循环	121
4.3.4 多重循环	122
4.3.5 串操作指令	124
习题 4	129
第 5 章 模块化程序设计	133
5.1 子程序结构	133
5.1.1 子程序指令	133
5.1.2 子程序设计	137
5.2 参数传递	138
5.2.1 寄存器传递参数	138
5.2.2 共享变量传递参数	141
5.2.3 堆栈传递参数	145

5.3 多模块程序结构	148
5.3.1 源文件包含	148
5.3.2 模块连接	151
5.3.3 子程序库	152
5.4 宏结构	153
5.4.1 宏汇编	153
5.4.2 重复汇编	160
5.4.3 条件汇编	162
习题 5	165
第 6 章 32 位指令及其编程	169
6.1 Intel 80x86 处理器	169
6.1.1 16 位 80x86 处理器	169
6.1.2 IA-32 处理器	169
6.1.3 Intel 64 处理器	171
6.2 32 位指令运行环境	172
6.2.1 32 位寄存器	172
6.2.2 存储器模型	174
6.2.3 32 位寻址方式	176
6.2.4 32 位指令代码	178
6.3 32 位整数指令系统	179
6.3.1 32 位扩展指令	180
6.3.2 32 位新增指令	183
6.4 DOS 平台的 32 位指令编程	186
习题 6	190
第 7 章 Windows 编程	192
7.1 操作系统函数调用	192
7.1.1 动态链接库	192
7.1.2 MASM 的过程声明和调用	193
7.1.3 程序退出函数	194
7.1.4 Windows 程序格式	195
7.2 控制台应用程序	196
7.2.1 控制台输出	196
7.2.2 控制台输入	199
7.3 图形窗口应用程序	202
7.3.1 消息窗口	202
7.3.2 结构变量	203
7.3.3 MASM 的高级语言特性	206
7.3.4 简单窗口程序	214

习题 7	222
第 8 章 与 Visual C++ 的混合编程	225
8.1 嵌入汇编	225
8.2 模块连接	229
8.2.1 约定规则	229
8.2.2 堆栈帧	231
8.3 调用高级语言函数	238
8.3.1 嵌入汇编程序中调用高级语言函数	239
8.3.2 汇编程序中调用 C 库函数	239
8.4 使用 Visual C++ 开发环境	240
8.4.1 汇编语言程序的开发过程	241
8.4.2 汇编程序的调试过程	242
习题 8	245
第 9 章 浮点、多媒体及 64 位指令	249
9.1 浮点指令	249
9.1.1 实数编码	250
9.1.2 浮点寄存器	253
9.1.3 浮点指令编程	256
9.2 多媒体指令	259
9.2.1 MMX	260
9.2.2 SSE	262
9.2.3 SSE2	264
9.2.4 SSE3	265
9.3 64 位指令	266
9.3.1 64 位方式的运行环境	267
9.3.2 64 位方式的指令	268
习题 9	269
附录 A 调试程序 DEBUG	272
A.1 DEBUG 程序的调用	272
A.2 DEBUG 程序的命令	272
A.3 DEBUG 程序的使用	277
附录 B 常用 DOS 功能调用	280
附录 C 输入输出子程序库	281
附录 D 列表文件符号说明	283
附录 E 常见汇编错误信息	284
附录 F 通用指令列表	286
附录 G MASM 伪指令和操作符列表	290
参考文献	291

第 1 章 汇编语言基础

程序设计语言是人与计算机沟通的语言，程序员利用它进行软件开发。通常人们习惯使用类似自然语言的高级程序设计语言，例如 C、C++ 或者 Basic、Java 语言等。高级语言需要翻译为计算机能够识别的指令，即机器语言，才能被计算机直接执行。机器语言是一串由 0 和 1 组成的二进制代码，对程序员来说艰涩难懂，被称为低级语言。将二进制代码的指令和数据用便于记忆的符号——助记符——表示就形成汇编语言（Assembly），所以汇编语言是一种面向机器的低级程序设计语言，或称为低层语言。

本章首先理解汇编语言的硬件基础：个人计算机、处理器的寄存器和存储器组织，然后学习汇编语言的程序格式，接着编写出第一个汇编语言程序，最后掌握基于 DOS 操作系统和微软 MASM 汇编程序的程序开发方法。

1.1 个人计算机系统概述

计算机系统包括硬件和软件两大部分。硬件（Hardware）是指构成计算机的实在的物理设备，是我们看得见、摸得着的物体，就像人的躯体。软件（Software）一般是指在计算机上运行的程序（广义的软件还包括由计算机管理的数据以及有关的文档资料），是指示计算机工作的命令，就像人的思想。

微型计算机（Microcomputer），简称微机，是最常使用的一类计算机，在科学计算、信息管理、自动控制、人工智能等领域有着广泛的应用。工作、学习和娱乐中使用的个人计算机（PC）是我们最熟悉、也是最典型的通用微型计算机。

1.1.1 计算机的硬件

源于冯·诺依曼设计思想的计算机由 5 大部件组成：控制器、运算器、存储器、输入设备和输出设备。控制器是整个计算机的控制核心；运算器是对数据进行运算处理的部件；存储器是用来存放数据和程序的部件；输入设备将数据和程序变换成计算机内部所能识别和接受的信息方式，并把它们送入存储器中；输出设备将计算机处理的结果以人们能接受的或其他机器能接受的形式送出。

现代计算机在很多方面都对冯·诺依曼计算机结构进行了改进，5 大部件演变为 3 个硬件子系统：处理器、存储系统和输入输出系统。运算器和控制器被制作在一块大规模集成电路芯片上，称为处理器（Processor），也常被称为中央处理单元 CPU（Central Processing Unit）。传统的存储器也发展成为存储系统，由寄存器、高速缓冲存储器、主存储器及辅助存储器构成。处理器和存储系统在信息处理中起主要作用，是计算机硬件的主体部分，通常被称为“主机”。输入设备和输出设备统称为外部设备，简称为外设或 I/O 设备；输入输出系统的主体是外部设备，但还包括外设与主机之间相互连接的 I/O 接口电路。

为简化各个部件的相互连接，现代计算机广泛应用总线结构，参见图 1-1。采用总线连接系统中各个功能部件使得计算机系统具有了组合灵活、扩展方便的特点。

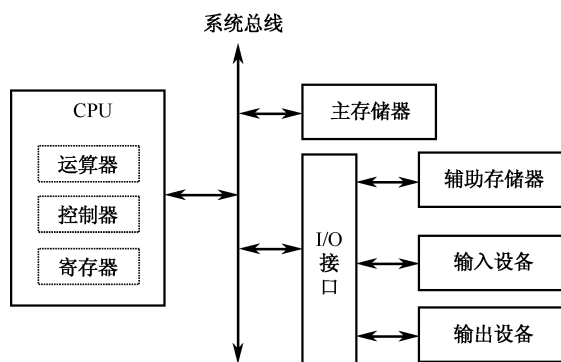


图 1-1 计算机系统的硬件组成

1. 处理器

处理器是计算机的运算和控制核心，微机中可被称为微处理器（Microprocessor）。现代通用微处理器功能非常强大，人们已经习惯称之为处理器或者 CPU。

（微）处理器芯片内集成了控制器、运算器和若干高速存储单元（即寄存器）。高性能处理器内部非常复杂，例如运算器中不仅有基本的整数运算器、还有浮点处理单元甚至多媒体数据运算单元，控制器还会包括存储管理单元、代码保护机制等，为提高存储器的性能还会集成高速缓冲存储器。处理器及其支持电路构成了计算机系统的处理和控制中心，对系统的各个部件进行统一的协调和控制。

PC 采用美国英特尔（Intel）公司的 80x86 系列处理器或与其兼容的处理器，例如常用的奔腾系列处理器或者酷睿系列多核处理器。之所以称之为 Intel 80x86 系列处理器，是因为它们都源于 16 位结构的 Intel 8086 处理器，而 8086 具有的所有指令，即指令系统是整个 Intel 80x86 系列处理器的基本指令集。

本书前 5 章将介绍基于 8086 处理器的 16 位常用指令，从第 6 章开始介绍 32 位指令。

2. 存储器

存储器（Memory）是计算机的记忆部件，存放程序和数据。存储系统由处理器内部的寄存器（Register）、高速缓冲存储器（Cache）、主板上的主存储器和以外设形式出现的辅助存储器构成。

按所起作用，存储器可分为主存储器和辅助存储器。主存储器（简称主存或内存）由半导体存储器芯片组成，安装在机器内部的电路板上，相对辅助存储器来说速度快，但容量小：造价高，主要用来存放当前正在运行的程序和等待处理的数据。辅助存储器（简称辅存或外存）主要由磁盘、光盘存储器等构成，以外设的形式安装在机器上，相对主存储器来说造价低、容量大、信息可长期保存，但速度慢，主要用来长久保存程序和数据。一般来说，程序和数据以文件形式保存在辅存上，只有使用它们时才读入主存。

按读写功能，存储器可分为可读可写存储器和只读存储器 ROM（Read Only Memory）。半导体存储器具有按指定位置访问，即随机存取的特点，所以可读可写的半导体存储器常被称为 RAM（Random Access Memory）。构成主存既需要 RAM，也需要 ROM，但需要注意的是，存放在 RAM 芯片上的信息断电后将会丢失，而 ROM 芯片中的信息则可在断电后保存。通常作为辅存的磁盘存储器和 CD-ROM 光盘都可以在断电后长期保存信息，它们二者的不

同在于，CD-ROM 光盘是只读的，而作为辅存的磁盘存储器是可读可写的。不过，由于读写时涉及磁头或光头的移动、磁盘或光盘的旋转，它们的存取性能低于半导体存储器。

个人计算机的主存由半导体存储芯片 RAM 和 ROM 构成。在 16 位 PC 系列机时代，RAM 容量不过是 64KB 或 1MB。32 位 PC 的 RAM 容量从最初的 4MB，逐渐发展直到 2010 年的 2GB 或 4GB。由于大量应用程序都需要 RAM 主存空间，因此 PC 的主存主要由 RAM 构成，俗称主存条（内存条）。

个人计算机的 ROM 部分主要是固化的 ROM-BIOS。BIOS（Basic Input/Output System）表示“基本输入输出系统”，是 PC 软件系统最底层的程序。它由诸多子程序组成，主要用于驱动和管理诸如键盘、显示器、打印机、磁盘、时钟、串行通信接口等基本的输入输出设备。操作系统通过对 BIOS 的调用驱动各硬件设备，用户也可以在应用程序中调用 BIOS 中的许多功能。ROM 空间还包含机器复位后初始化系统的程序，它将操作系统引导到 RAM 存储器中执行。

3. 外部设备

外部设备是指计算机上配备的输入（Input）设备和输出（Output）设备，也称 I/O 设备或外围设备，简称外设（Peripheral），其作用是让用户与计算机实现交互。

个人计算机上配置的标准输入设备是键盘、标准输出设备是显示器，二者又合称为控制台（Console）。个人计算机还可使用鼠标、打印机等 I/O 设备。作为外部存储器驱动装置的磁盘驱动器，既是输出设备，又是输入设备。

由于各种外设的工作速度、驱动方法差别很大，无法与处理器直接匹配，所以不可能将它们直接连接到主机。这里就需要有一个 I/O 接口来充当外设和主机之间的桥梁，通过该接口电路来完成信号变换、数据缓冲、联络控制等工作。在个人计算机中，较复杂的 I/O 接口电路通常制成独立的电路板，也常被称为接口卡（Card），例如显示卡，使用时需要将其插在主板的总线插槽上。

4. 系统总线

总线（Bus）是用于多个部件相互连接、传递信息的公共通道，物理上就是一组公用导线。例如，处理器芯片的对外引脚（Pin）常被称为处理器总线。这里的系统总线（System Bus）是指计算机系统中主要的总线，例如处理器与存储器和 I/O 设备进行信息交换的公共通道。

16 位 PC 采用 16 位工业标准结构 ISA（Industry Standard Architecture）系统总线连接各个功能部件。32 位 PC 上使用外设部件互连 PCI（Peripheral Component Interconnect）总线连接 I/O 接口卡。系统总线除了作为主机板上处理器、主存和 I/O 接口的公共通道外，主板上还设置有许多系统总线插槽，主要用于插接 I/O 接口电路以扩充系统连接的外设，故也被称作 I/O 通道。

对汇编语言程序员来说，处理器、存储器和外部设备依次被抽象为寄存器、存储器地址和输入输出地址，因为编程过程中将只能通过寄存器和地址实现处理器控制、存储器和外设的数据存取及处理等操作。

1.1.2 计算机的软件

完整的计算机系统包括硬件和软件，软件又分为系统软件和应用软件。

1. 系统软件

系统软件是为了方便使用、维护和管理计算机系统的程序及其文档，通常由计算机生产厂商或者软件公司提供，其中最重要的是操作系统。

操作系统 OS (Operating System) 管理着系统的软硬件资源，为用户提供使用机器的交互界面，为程序员使用资源提供可供调用的驱动程序，为其他程序构建稳定的运行平台。

程序员采用某种程序设计语言编写源程序，利用语言翻译程序将源程序转变成可运行的程序。例如，本书介绍用汇编语言编写源程序的方法，但源程序必须利用“汇编程序”完成翻译才可被计算机执行。高级语言则采用编译类或解释类程序来完成这个工作。辅助程序开发的程序设计语言的翻译程序等一般也归类为系统软件。

2. 应用软件

应用软件是解决某个问题的程序及其文档，大到用于处理某专业领域问题的程序，小到完成一个非常具体工作的程序。它覆盖了计算机应用的所有方面，每个应用都需要相应的应用程序。

个人计算机系统具有多种多样的应用软件。例如，进行程序设计时要采用文本编辑软件编写源程序，带有丰富格式的字处理软件帮助你书写文章，排版软件则用于书刊出版。

大型的程序设计项目往往要借助软件开发工具（包）。这个开发工具是进行程序设计所用的各种程序的有机集合，所以也被称为集成开发环境，包括文本编辑器、语言翻译程序，有用于形成可执行文件的连接程序，以及进行程序排错的调试程序等。

1.1.3 程序设计语言

利用计算机解决实际问题，一般要编制程序。程序设计语言就是程序员用来编写程序的语言，它是人与计算机之间交流的工具。程序设计语言可以分为机器语言、汇编语言和高级语言。

1. 机器语言

计算机能够直接识别的是由二进制数 0 和 1 组成的代码。机器指令 (Instruction) 就是用二进制编码的指令，指令是控制计算机操作的命令，是处理器不需要翻译就能识别（直接执行）的“母语”，通常一条机器指令控制计算机完成一个操作。每种处理器都有各自的机器指令，某处理器支持的所有指令的集合就是该处理器的指令集 (Instruction Set) 或指令系统。指令集及使用它们编写程序的规则被称作机器语言 (Machine Language)。

用机器语言形成的程序是计算机唯一能够直接识别、并执行的程序，而用其他语言编写的程序必须经过翻译、变换成机器语言程序；所以，机器语言程序常称为目标程序（或目的程序）。

例如，完成两个数据 100 和 256 相加的功能，在 8086 处理器的二进制代码序列如下：

```
10111000 01100100 00000000  
00000101 00000000 00000001
```

几乎没有人能够直接读懂该程序段的功能，因为机器语言就是看起来毫无意义的一串代码。用机器语言编写程序的最大缺点是难以理解，因而极易出错，也难以发现错误。所以，只是在计算机发展的早期或不得已的情况下，才采用机器语言编写程序。现在，除了有时在

程序某处需要直接采用机器指令填充外，几乎没有人采用机器语言编写程序了。

二进制虽然只有 0 和 1 两个数码，但表达信息时会很长。为了简化表达，常用到十六进制。因为一个十六进制位就可以表达 4 位二进制数，并且易于相互转换，即二进制数 0000、0001、0010、……、1001、1010、1011、1100、1110、1111 用十六进制表达依次是 0、1、2、……、9、A、B、C、D、E、F，其中 A~F 依次表示十进制 10~15。这样，上述二进制代码序列用十六进制代码表示为：

B8 64 00

05 00 01

汇编语言中，习惯使用后缀字母区别不同进制的数据。例如，使用字母 B（或小写字母形式 b，来自二进制的英文单词 Binary）表示二进制数，使用字母 H（或小写字母形式 h，来自十六进制的英文单词 Hexadecimal）表示十六进制数，而十进制数通常不需要特别说明（或者用后缀字母 D 或 d，以示强调）。另外，涉及计算机硬件原理的技术文献中，所谓的“位”常指二进制位，也会表示十六进制位、或者十进制位，根据上下文予以分辨，否则可能产生误解。

2. 汇编语言

为了克服机器语言的缺点，人们采用便于记忆并能描述指令功能的符号来表示机器指令。表示指令功能的符号称为指令助记符，或简称助记符（Mnemonic）；助记符一般采用表明指令功能的英语单词或其缩写。指令操作数同样也可以用易于记忆的符号表示。用助记符表示的指令就是汇编格式指令。汇编格式指令以及使用它们编写程序的规则形成汇编语言（Assembly Language）。用汇编语言书写的程序就是汇编语言程序，或称汇编语言源程序。

例如，实现 100 与 256 相加的 MASM 汇编语言程序片段如下：

```
mov ax,100
```

```
add ax,256
```

第一条指令的功能将数据 100 传送给名为 AX 的寄存器，MOV 是传送指令的助记符，实现赋值功能。该指令对应的机器代码就是机器语言举例的第一个二进制串。

第二条指令实现加法操作，ADD 是加法指令的助记符，对应机器语言举例的第二个二进制串。

如果熟悉有关助记符及对应指令的功能，就可以读懂上述程序片段了。

汇编语言是一种符号语言，它用助记符表示操作码，比机器语言容易理解和掌握、也容易调试和维护。但是，汇编语言源程序要翻译成机器语言程序才可以由处理器执行。这个翻译的过程称为“汇编”，完成汇编工作的程序就是汇编程序（Assembler）。

3. 高级语言

汇编语言虽然较机器语言直观一些，但仍然烦琐难记。于是在 20 世纪 50 年代，人们研制出了高级程序设计语言（High Level Language）。高级语言比较接近于人类自然语言的语法习惯及数学表达形式，它与具体的计算机硬件无关，更容易被广大计算机工作者掌握和使用。利用高级语言，即使一般的计算机用户也可以编写程序，而不必懂得计算机的结构和工作原理。

目前，计算机高级语言已有上百种之多，得到广泛应用的有十几种，每种高级语言都有其最适用的领域。用任何一种高级语言编写的程序都要通过编译程序（Compiler）翻译成机器语言程序（称为目标程序）后计算机才能执行，或者通过解释程序边解释边执行。

例如，用高级语言表达 100 与 256 相加，就是通常的数学表达形式： $100 + 256$ 。

4. 学习汇编语言的意义

高级语言简单、易用，而汇编语言复杂、难懂，是否就没有必要再采用汇编语言了呢？让我们首先列表 1-1 简单比较一下汇编语言和高级语言的特点。

表 1-1 汇编语言和高级语言的对比

汇编语言	高级语言
与处理器密切相关。每种处理器都有自己的指令系统，相应的汇编语言各不相同。所以，汇编语言程序的通用性、可移植性较差	与具体计算机无关，高级语言程序可以在多种计算机上编译后执行
功能有限，又涉及寄存器、主存单元等硬件细节。所以，编写程序比较烦琐，调试起来也比较困难	采用类似自然语言的语法，不必关心诸如标志、堆栈等琐碎问题。容易被掌握和应用
本质上就是机器语言，可以直接、有效地控制计算机硬件，因而容易产生运行速度快、指令序列短小的高效率目标程序	不易直接控制计算机的各种操作，编译程序产生的目标程序往往比较庞大，运行速度较慢

通过对比，高级语言的优势明显。很自然地人们称机器语言和汇编语言为低级语言。但事实上，汇编语言被称为低层语言（Low Level Language）更合适。因为，程序设计语言是按照计算机系统的层次结构区分的，本没有“高低贵贱”之分，只是某种语言更适合某种应用层面（或说场合）而已。我们看到，汇编语言便于直接控制计算机硬件电路，可以编写在“时间”和“空间”两方面最有效，即执行速度快和目标代码小的程序。这些优点使得汇编语言在程序设计中占有重要的位置，是不可被取代的。下面罗列了汇编语言的主要应用场合：

- ④ 程序要具有较快的执行时间，或者只能占用较小的存储容量。例如，操作系统的核心程序段，实时控制系统的软件，智能仪器仪表的控制程序等。
- ④ 程序与计算机硬件密切相关，程序要直接、有效地控制硬件。例如，I/O 接口电路的初始化程序段，外部设备的低层驱动程序等。
- ④ 大型软件需要提高性能、优化处理的部分。例如，计算机系统频繁调用的子程序、动态连接库等。
- ④ 没有合适的高级语言、或只能采用汇编语言的时候。例如，开发最新的处理器程序时，暂时没有支持新指令的编译程序。
- ④ 许多实际应用的情况，例如分析具体系统尤其是该系统的低层软件、加密解密软件、分析和防治计算机病毒等。

当然，无须回避的事实是，随着各种编程技术的发展，单独使用汇编语言开发程序、尤其是应用程序的情况越来越少。所以，在实际的程序开发过程中，可以采用高级语言和汇编语言混合编程的方法，互相取长补短，更好地解决实际问题。

另外，编写汇编语言程序，需要使用处理器指令解决应用问题，而指令只是完成诸如将一个数据从存储器传送到寄存器、对两个寄存器值求和、指针增量指向下一个地址等简单的功能。所以，从教学角度来说，汇编语言程序员在将复杂的应用问题翻译成简单指令的过程中，就是从处理器角度解决问题，自然就容易理解计算机的工作原理了。

1.2 8086 处理器

处理器是计算机系统的硬件核心部件，决定着计算机的关键性能，常被称为中央处理单元，简称 CPU。了解处理器的基本结构、熟悉其寄存器作用以及存储器的组织是学习处理器指令的基础。

1.2.1 8086 的功能结构

处理器由多个功能部件组成。Intel 公司按两个功能模块描绘了 Intel 8086 处理器的内部结构，如图 1-2 所示。

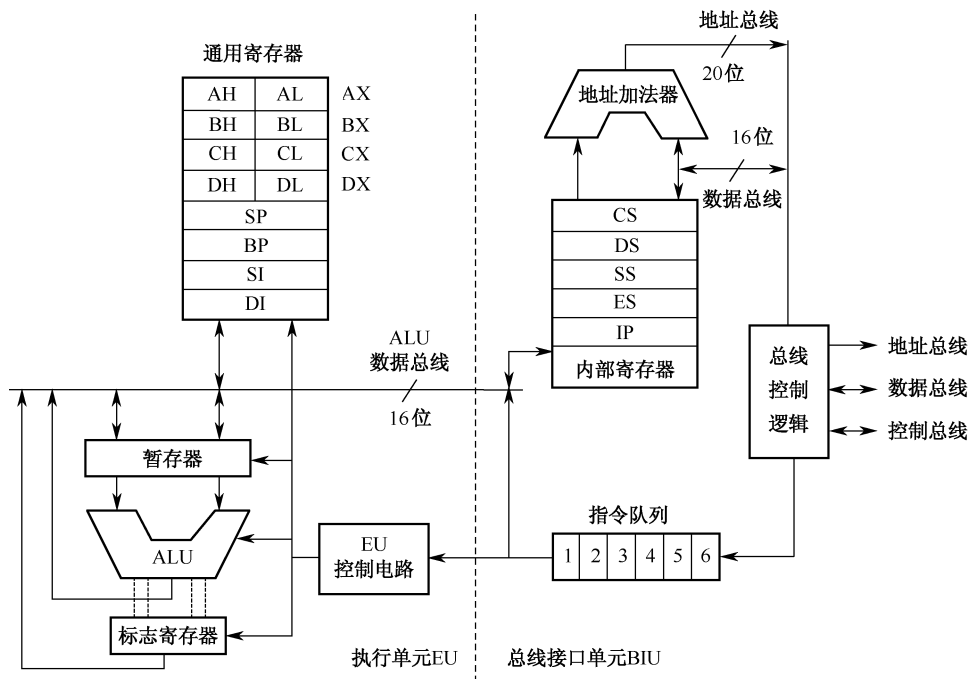


图 1-2 8086 的内部结构

1. 总线接口单元和执行单元

图 1-2 虚线右边部分是总线接口单元 BIU (Bus Interface Unit)。它由 6 字节的指令队列 (即指令寄存器)、指令指针 IP (等同于程序计数器的功能)、段寄存器 (CS、DS、SS 和 ES)、地址加法器和总线控制逻辑等构成。该单元管理着 8086 与系统总线的接口，负责处理器对存储器和外设进行访问。对外的 8086 处理器引脚由 16 位双向数据总线、20 位地址总线和若干控制总线组成。8086 所有对外操作必须通过 BIU 和这些总线进行，例如从主存中读取指令、从主存或外设读取数据、向主存或外设写出数据等操作。

图 1-2 虚线左边部分是执行单元 EU (Execution Unit)。它由算术逻辑单元 ALU (Arithmetic Logic Unit)、标志寄存器、通用寄存器和进行指令译码的 EU 控制电路等构成，负责执行指令的功能。

源程序由一条条语句组成，而可执行程序则由一条条指令组成，运行程序就是执行一条条

指令的过程。一条指令的整个执行过程又可以分成两个主要阶段：取指和执行，如图 1-3（a）所示。

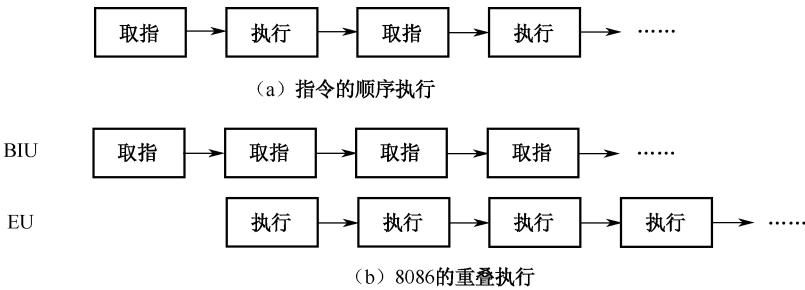


图 1-3 指令的执行过程

取指阶段是处理器将指令代码从主存储器中取出并进入处理器内部的过程。8086 处理器中，指令在存储器中的地址（即位置编号）由代码段寄存器 CS 和指令指针寄存器 IP 共同提供，再由地址加法器得到 20 位存储器地址。总线接口单元 BIU 负责从存储器取出这个指令代码，送入指令队列。

执行阶段是处理器将指令代码翻译成它代表的功能（被称为译码）、并发出有关控制信号实现这个过程。8086 处理器中，执行单元 EU 从指令队列中获得预先取出的指令代码，在 EU 控制电路中进行译码，然后发出控制信号由算术逻辑单元 ALU 进行数据运算、数据传送等操作。指令执行阶段需要的操作数据有些来自处理器内部的寄存器，有些来自指令队列，还有些来自存储器和外设。如果需要来自存储器或外设的数据，则控制单元 EU 控制总线接口单元 BIU 从外部获取这些数据。

2. 指令预取

8086 处理器维护着长度为 6 字节的指令队列，该队列按照“先进先出”FIFO（First In First Out）的方式进行工作。当指令队列中出现空缺时，BIU 会自动取指填补这一空缺；而当程序不能按顺序执行，即发生转移（出现分支）时，BIU 又会废除已经取出的指令，重新取指形成新的指令队列。

8086 处理器中，指令的读取操作在 BIU 单元实现，而指令的执行阶段在 EU 单元完成。因为 BIU 和 EU 两个单元相互独立、分别完成各自操作，所以可以并行操作。换句话说，也就是在 EU 单元对一个指令进行译码执行时，BIU 单元可以同时对外部指令进行读取；所以，8086 处理器的指令读取，实际上是指令预取（Prefetch），如图 1-3（b）所示。

由于要译码执行的指令已经预取到了处理器内部的指令队列，所以 8086 不需要等待取指操作就可以从指令队列获得指令进行译码执行。而对于简单的处理器来说，在指令译码前必须等待取指操作的完成。取指是处理器最频繁的操作，每条指令都要读取指令代码一到数次（与指令代码的长度有关），所以 8086 的这种结构和操作方式节省了处理器的许多取指时间，提高了工作效率。这就是最简单的指令流水线技术。同时也看到，程序转移将使预取指令作废，从而降低了流水线效率。

1.2.2 8086 的寄存器

处理器内部需要高速存储单元，用于暂时存放程序执行过程中的代码和数据，这些存储

单元被称为寄存器（Register）。处理器内部设计有多种寄存器，每种寄存器还可能有多，从应用的角度可以分成两类：透明寄存器和可编程寄存器。

有些寄存器对应用人员来说不能通过指令直接编程控制，例如：保存指令代码的指令寄存器。它们对应用人员来说好像看不见一样，被称为透明寄存器。这里的“透明”（Transparency）是计算机学科中常用的一个专业术语，表示实际存在但从某个角度看好像没有；运用“透明”思想可以使我们抛开不必要的细节，而专注于关键问题。

底层语言程序员需要掌握可编程（Programmable）寄存器。它们具有引用名称、供编程使用，还可以进一步分成通用和专用寄存器：

- ④ 通用寄存器：这类寄存器在处理器中数量较多、使用频度较高，具有多种用途。例如它们可用来存放指令需要的操作数据，又可用来存放地址以便在主存或 I/O 接口中指定操作数据的位置。
- ④ 专用寄存器：这类寄存器各自只用于特定目的。例如程序计数器 PC（Program Counter）只用于记录将要执行指令的主存地址，标志寄存器保存指令执行的辅助信息。

8086 的寄存器有 8 个 16 位通用寄存器、4 个 16 位段寄存器、1 个 16 位标志寄存器和 1 个 16 位指令指针寄存器，如图 1-4（图中数字 15、7、0 等依次用于表达二进制位 D15、D7、D0）所示。

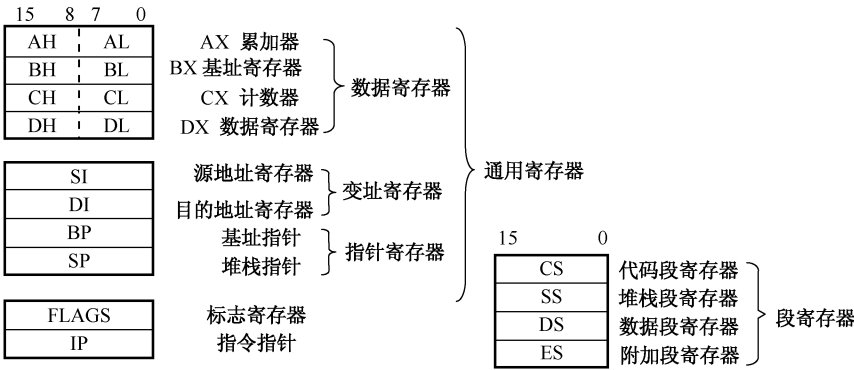


图 1-4 8086 的寄存器

1. 通用寄存器

通用寄存器（General-Purpose Register）一般是指处理器最常使用的整数通用寄存器，可用于保存整数数据、地址等。8086 处理器只有 8 个通用寄存器，数量有限。

8086 处理器的 8 个 16 位通用寄存器，分别被命名为：AX、BX、CX、DX、SI、DI、BP 和 SP。其中前 4 个通用寄存器 AX、BX、CX 和 DX 还可以进一步分成高字节 H（High）和低字节 L（Low）两部分，这样又有了 8 个 8 位通用寄存器：AH 和 AL、BH 和 BL、CH 和 CL、DH 和 DL。前 4 个通用寄存器在编程中，可以整个使用 16 位寄存器（例如：AX），也可以分成两个 8 位使用：D15~D8（例如：AH）和 D7~D0（例如：AL），对其中某 8 位的操作，并不影响另外对应 8 位的数据。

通用寄存器的用途很多，可以保存数据、暂存运算结果，也可以存放存储器地址、作为变量的指针。但在 8086 处理器中每个寄存器又有它们各自的特定作用，并因而得名。程序中通常也按照其含义使用它们，如表 1-2 所示。

表 1-2 8086 处理器的通用寄存器

名 称	中英文含义	作 用
AX	累加器 (Accumulator)	使用频度最高, 用于算术、逻辑运算以及与外设传送信息等
BX	基址寄存器 (Base)	常用做存放存储器地址, 以方便指向变量或数组中的元素
CX	计数器 (Counter)	常作为循环操作等指令中的计数器
DX	数据寄存器 (Data)	可用来存放数据, 在输入输出指令存放外设端口地址
SI	源变址寄存器 (Source Index)	用于指向字符串或数组的源操作数
DI	目的变址寄存器 (Destination Index)	用于指向字符串或数组的目的操作数
BP	基址指针寄存器 (Base Pointer)	默认情况下指向程序堆栈区域的数据, 主要用于在子程序中访问通过堆栈传递的参数和局部变量
SP	堆栈指针寄存器 (Stack Pointer)	专用于指向程序堆栈区域顶部的数据, 在涉及堆栈操作的指令中会自动增加或减少

许多指令需要表达两个操作数 (操作对象, 例如加法指令的被加数以及加法结果):

- ⊙ 源操作数是指被传送或参与运算的操作数 (例如: 加法的被加数)。
- ⊙ 目的操作数是指保存传送结果或运算结果的操作数 (例如: 加法的和值结果)。

SI 和 DI 是变址寄存器, 常通过改变寄存器表达的地址指向数组元素。SI 常用于指向源操作数, 而 DI 常用于指向目的操作数。

堆栈 (Stack) 是一个特殊的存储区域, 它采用先进后出 FILO (First In Last Out)、也称为后进先出 LIFO (Last In First Out) 的操作方式存取数据。它用于调用子程序时暂存数据、传递参数、存放局部变量, 也可以用于临时保存数据。BP 和 SP 是指针寄存器, 用于指向堆栈中的数据。其中, SP 堆栈指针会随着处理器执行有关指令自动增大或减小, 所以 SP 不应该再用于其他目的, 实际上可归类为专用寄存器; 但是 SP 又可以像其他通用寄存器一样灵活地改变。

2. 标志寄存器

标志 (Flag) 用于反映指令执行结果或控制指令执行形式。许多指令执行之后将影响有关的状态标志位; 不少指令的执行要利用某些标志; 当然, 也有很多指令与标志无关。处理器中用一个或多个二进制位表示一种标志, 其 0 或 1 的不同组合表达标志的不同状态。Intel 8086 支持的 9 个标志, 分为状态标志和控制标志两类, 采用一个 16 位的标志寄存器 FLAGS 保存, 如图 1-5 所示 (图上方的数字表示该标志在标志寄存器中的位置)。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

图 1-5 标志寄存器 FLAGS

状态标志是最基本的标志, 用来记录指令执行结果的辅助信息。加减运算和逻辑运算指令是主要设置它们的指令, 其他有些指令的执行也会相应地设置它们。状态标志有 6 个, 处理器主要使用其中 5 个构成各种条件, 分支指令判断这些条件实现程序分支。它们从低位到高位是: 进位标志 CF (Carry Flag)、奇偶标志 PF (Parity Flag)、调整标志 AF (Adjust Flag)、零标志 ZF (Zero Flag)、符号标志 SF (Sign Flag)、溢出标志 OF (Overflow Flag)。

控制标志用于控制处理器执行指令的方式, 可由程序根据需要用相关指令设置。8086 的控制标志有 3 个: 方向标志 DF (Direction Flag), 仅用于串操作指令中, 控制地址的变化方

向；中断允许标志 IF (Interrupt-enable Flag)，或简称中断标志，用于控制外部可屏蔽中断是否可以被处理器响应；陷阱标志 TF (Trap Flag)，也常称为单步标志，用于控制处理器是否进入单步操作方式。

3. 指令指针寄存器

程序由指令组成，指令存放在主存储器中。处理器需要一个专用寄存器表示将要执行的指令在主存的位置，这个位置用存储器地址表示。在 8086 处理器中，这个存储器地址保存在 16 位指令指针寄存器 IP (Instruction Pointer) 中。

指令指针寄存器 IP 一个是专用寄存器，具有自动增量的能力。处理器执行完一条指令，IP 就加上该指令的字节数，指向下一条指令，实现程序的顺序执行。需要实现分支、调用等操作时需要修改 IP，它的改变将引起程序转移到指定的指令执行。但 IP 寄存器不能像通用寄存器那样直接赋值修改，需要执行控制转移指令（如跳转、分支、调用和返回指令）、出现中断或异常时被处理器赋值而相应改变。

4. 段寄存器

一个程序当中，有可以执行的指令代码，还有指令操作的各类数据等。遵循模块化程序设计思想，希望将相关的代码安排在一起，相关的数据安排在一起，于是（区）段 (Segment) 的概念自然出现。一个段安排一类代码或数据。程序员在编写程序时，可以很自然地把程序的各部分放在相应的段中。对应用程序来说，主要涉及 3 类基本段：存放程序中指令代码的代码段 (Code Segment)、存放当前运行程序所用数据的数据段 (Data Segment) 和指明程序使用的堆栈区域的堆栈段 (Stack Segment)。

段其实就是主存的一个连续区域，为了表明段在主存中的位置，8086 处理器设计有 4 个 16 位段寄存器：代码段寄存器 CS，堆栈段寄存器 SS，数据段寄存器 DS 和附加段寄存器 ES (Extra Segment)。其中，附加段也是用于存放数据的数据段，专为处理数据串设计的串操作指令必须使用附加段作为其目的操作数的存放区域。

1.2.3 8086 的存储器组织

个人计算机主板上的主存条和一个 ROM 芯片构成主存储器，保存正在运行使用的指令和数据。处理器从存储器读取指令，在执行指令的过程中读写数据。对存储器的基本操作是按照要求向指定地址（位置）存进（即写入 Write）或取出（即读出 Read）信息。应该注意的是，存储器内容不会因为被读出而消失，所以更准确地说是获取其副本，而且可以重复取出；只有存入新的信息后，原有信息才会被更改。

1. 存储单元和存储单位

主存储器是一个很大的信息储存库，被划分成许多存储单元。为了区分和识别各个存储单元，并按指定位置进行存取，就给每个存储单元编排一个顺序号码，称为存储单元地址或存储器地址 (Memory Address)。现代计算机中，主存储器的每个存储单元具有一个地址，保存一个字节（8 个二进制位）的信息，这称为字节编址 (Byte Addressable)，也直译为字节可寻址，因为通过一个存储器地址可以访问到一个字节信息。

计算机中信息的基本单位是一个二进制位 (bit，中文译为比特)，一般使用小写字母 b

表示，一个二进制位可表示一位二进制数：0 或 1。8 个二进制位组成一个字节（Byte），常用大写字母 B 表示，位编号由右向左（由低位向高位）从 0 开始递增计数为 D0~D7，如图 1-6 所示。8086 是 16 位结构的处理器（即字长为 16 位），主要处理的数据长度是 16 位；所以称 16 个二进制位为一个字（Word），位编号自右向左为 D0~D15。一个 16 位字由 2 个字节组成。32 位数据由 4 个字节组成，称为双字（Double Word），位编号自右向左为 D0~D31。

一个二进制数据的右边最低位称为最低有效位 LSB（Least Significant Bit），即 D0 位；左边最高位称为最高有效位 MSB（Most Significant Bit），对应字节、字、双字长度的数据依次指 D7、D15 和 D31 位。



图 1-6 数据的位格式

主存储器使用字节作为基本存储单位，表 1-3 罗列了表达更大的存储容量时常使用的单位及关系。虽然借用了日常生活中千、兆、吉等单位，但没有使用其 10^3 的十进制倍数关系，而是使用 $2^{10}=1024$ 的二进制倍数关系（近似等于 $10^3=1000$ ）。但有些产品，例如硬盘、U 盘的生产厂商给出容量却采用 10^3 倍数关系，注意分辨。

表 1-3 常用存储单位

英文符号	中英文名称	二进制倍数关系	十进制倍数关系
K	千（Kilo）	1KB=2 ¹⁰ B=1024 B	1K=10 ³ B
M	兆（Mega）	1MB=2 ¹⁰ KB=2 ¹⁰ B	1M=10 ⁶ K
G	千兆，吉（Giga）	1GB=2 ¹⁰ MB=2 ³⁰ B	1G=10 ⁹ M
T	兆兆，太（Tera）	1TB=2 ¹⁰ GB=2 ⁴⁰ B	1T=10 ¹² G

2. 物理地址和逻辑地址

主存条和 ROM 芯片构成的主存储器需要处理器通过总线进行访问，被称为物理存储器。物理存储器的每个存储单元有一个唯一的地址，就是物理地址（Physical Address）。物理地址空间从 0 开始顺序编排，直到处理器支持的最大存储单元。

我们知道一个二进制位有 0 和 1 两种状态，即 2（=2¹）个编码；那么，2 位有 00、01、10 和 11 共 4（2²）个编码，以此类推，借助排列组合知识，可以得到 N 位有 2^N 个编码。计算机使用数字信号传输信息，一个数字信号也只有低电平和高电平两个状态，可以分别用 0 和 1 表达，所以 N 位数字信号同样也有 2^N 个不同的编码。如果用 N 个数字信号传输访问存储器的地址信息，就可以区别出 2^N 个不同的存储单元，也就是说可以直接访问到 2^N 个存储单元。

8086 处理器具有 20 位地址总线，即用 20 个数字信号访问存储器，故 8086 可以支持 2²⁰，即 1M 个存储单元。由于每个存储单元保存一个字节数据，所以 8086 可以支持 1MB 存储器容量，其物理地址空间是 0~2²⁰-1。地址（编号）习惯用十六进制数表达，20 位数字信号对应二进制 20 位，可以用 5 位十六进制数表达为：00000H~FFFFFH，参见图 1-7 左侧所示。

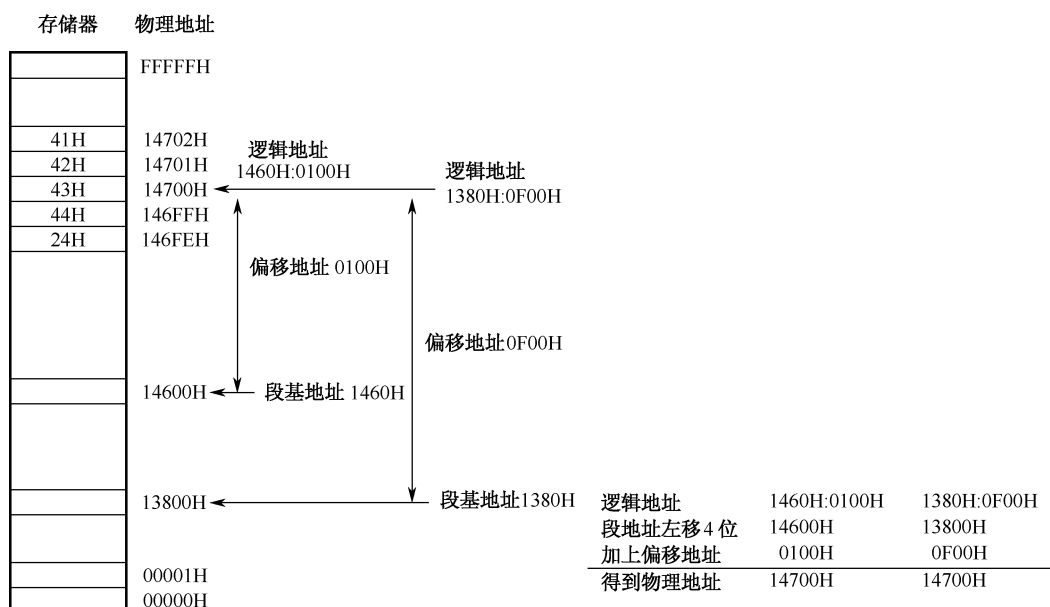


图 1-7 物理地址和逻辑地址

但是，8086 处理器只有 16 位寄存器，没有设计可以完整保存 20 位物理地址的寄存器。而使用二进制 16 位（对应十六进制 4 位）表示存储器地址，可以表达的存储器容量是 64KB ($=2^{16}$ 字节)，范围是：0000H~FFFFH。

于是，8086 处理器将 1MB 物理存储器空间分成许多不超过 64KB 的区域进行管理，这种区域被称为（区）段（Segment）。段用于 8086 内部和程序设计中，所以常被称为逻辑段。

由于规定每段最大 64KB，段内的存储单元就可以使用 16 位地址表示。同时，8086 处理器还规定每个段只能起始于低 4 位（二进制）全为 0 的物理地址位置，这种地址是模 16 地址，也就是可以被 16 整除的地址（用十进制表达是：0、16、 2×16 、 $3 \times 16 \dots$ ），用十六进制表达是：XXXX0H（X 表示十六进制一位，可以是 0~F 任何值）。既然段起始地址的低 4 位都是 0（对应十六进制是一位 0），可以省略不用表达，那这个 20 位物理地址就可以只表达出高 16 位。

这样，某个存储单元还可以通过它所在的（区）段及在该段中的位置指示，这就是在 8086 内部以及编程中使用的逻辑地址（Logical Address），它包括段基地址和偏移地址，都可以使用 16 位表示。段基地址（常简称段地址）确定该段在主存中的起始地址。以段基地址为起点，段内的位置可以用距离该起点的位移量表示，称为偏移地址（Offset）。

逻辑地址常借用 MASM 汇编程序的方法，使用英文冒号“:”分隔段基地址和偏移地址，形如“段基地址:偏移地址”。并且在 8086 中，段基地址常只表达高 16 位。例如逻辑地址“1460H:0100H”表示段基地址是 1460H，即该存储单元所在的段起始于物理地址 14600H；该存储单元的偏移地址是 0100H，说明它位于距离段起始位置的 0100H 处，参见图 1-7 中间所示。

编程使用的逻辑地址由 8086 总线接口单元 BIU 的地址加法器电路转换为物理地址，然后通过处理器引脚输出外部访问物理存储器。进行转换时，两个 16 位组成的逻辑地址中的段地址需要左移 4 位（十六进制一位），加上偏移地址才能得到 20 位物理地址。例如某存储单元的逻辑地址是“1460H:0100H”，其物理地址是 14700H，参见图 1-7 右边所示。

某个存储单元可以处于不同起点的逻辑段中（当然对应的偏移地址也就不同），所以可以有多个逻辑地址，但只有一个唯一的物理地址。或者说，同一个物理地址可以有多个逻辑地址。例如，物理地址 14700H 还可以用逻辑地址“1380H:0F00H”表示，该段起始于 13800H，参见图 1-7 所示。

3. 应用程序的基本段

8086 设计有 4 种类型的逻辑段实现存储器的分段管理，有 4 个段寄存器保存对应段的段基地址（注意，只是保存高 16 位），如表 1-4 所示。

表 1-4 8086 的 4 种逻辑段

段 名 称	段的作用	段基地址	偏移地址
代码段	存放程序的指令序列	CS	IP
堆栈段	确定堆栈所在的主存区域	SS	SP
数据段	存放当前运行程序所用的数据	DS	EA
附加段	附加的数据段，也用于保存数据	ES	EA

分段管理存储器空间也符合程序的模块化思想，应用程序通常需要使用 3 种类型的段：代码段、堆栈段和数据段，8086 还设计有一个属于数据段类型的附加段。

程序的指令代码必须安排在代码段，否则将无法正常运行。程序利用代码段寄存器 CS 获得当前代码段的段基地址，指令指针寄存器 IP 保存代码段中指令的偏移地址。处理器利用 CS:IP 取得下一条要执行的指令。

程序使用的堆栈（临时存放数据的区域）一定在堆栈段。程序利用 SS 获得当前堆栈段的段基地址，堆栈指针寄存器 SP 保存堆栈栈顶的偏移地址。处理器利用 SS:SP 操作堆栈数据。

一个程序可以使用多个数据段，便于安全有效地访问不同类型的数据。例如，程序的主要数据存放在一个数据段，只读的数据存放在另一个数据段，动态分配的数据安排在第 3 个数据段。8086 规定程序当前访问的数据默认安排在数据段（应用中无须特别说明），也允许将数据安排在其他段（需要明确指明），数据的偏移地址由各种存储器寻址方式计算出有效地址 EA（详见第 2 章）。

4. 存储器的分段管理

8086 规定段基地址低 4 位均为 0，每段最大不超过 64KB。但并没有要求每段必须是 64KB，各段之间可以完全分开，也可以部分重叠，甚至完全重叠。当然各段的内容是不允许发生冲突的。图 1-8 说明了这种情况。

图 1-8 (a) 是各段独立的分配示例。CS=0150H、DS=4200H、SS=1CD0H、ES=B000H，它们分别为代码段、数据段、堆栈段和附加段的段地址。自每个段地址开始，各段均占 64KB 字节的范围，各段之间互不重叠。

图 1-8 (b) 则是相互重叠段的分配示例。CS=0200H、DS=0400H、SS=0480H，这样代码段、数据段和堆栈段的物理起始地址分别为 02000H、04000H 和 04800H。其中代码段大小为 8KB，数据段占 2KB，而堆栈段只有 256 字节，SP=0100H。由于该程序没有使用附加段，所以没有设置 ES 值。从这里可以看出，各段大小应根据实际需要来分配，可以重叠。有时，甚至可以将所有 4 种段都集中在一个逻辑段内，形成一个短小紧凑的程序，其大小不超过 64KB。例如，在图 1-8 (b) 所示的情况下，如果设置 CS=DS=SS=0200H；这时，代

码段将占据该逻辑段从偏移地址 0000~1FFFH 的 8KB，而数据段在偏移地址 2000H~27FFH 位置，堆栈顶指针 SP=2900H。

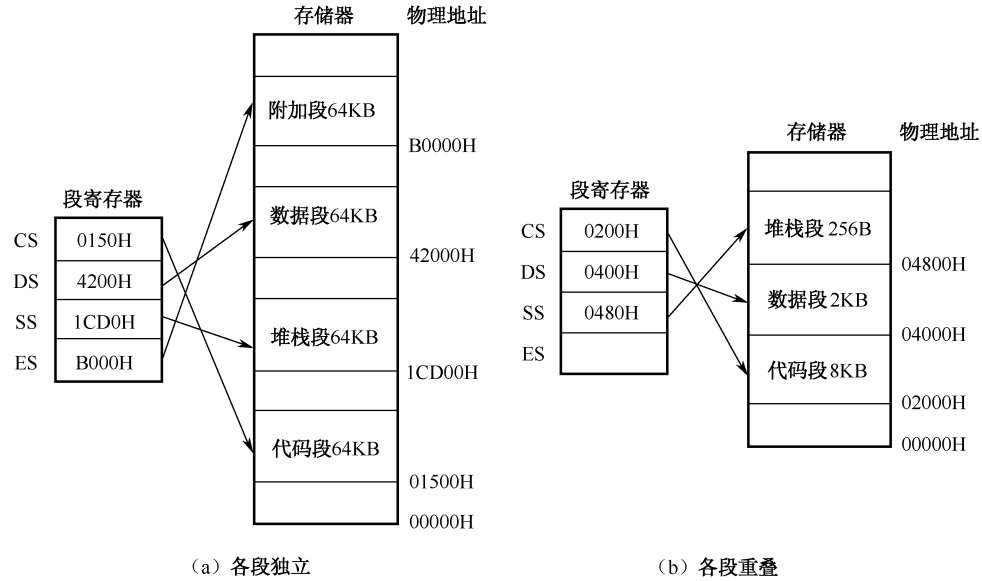


图 1-8 存储器的分段管理

1.3 汇编语言程序的格式

有了基本的硬件基础，下面开始学习汇编语言的语句格式和源程序框架，编写第一个汇编语言程序。

1.3.1 指令代码格式

汇编语言程序的主体是处理器指令，了解指令的代码格式有助于理解指令功能。

指令的代码格式（Instruction Format）说明如何用二进制编码指令，也称机器代码（Machine Code）格式，它由操作码和地址码组成。指令的操作码（Opcode）表明处理器执行的操作，例如数据传送、加法运算、跳转等操作。操作数（Operand）是参与操作的数据，也就是各种操作的对象，主要以寄存器或存储器地址、I/O 地址形式指明数据的来源，所以也称为地址码，例如，数据传送指令的源地址和目的地址，加法指令的加数、被加数及和值都是操作数。有些指令不需要操作数，通常的指令都有一个或两个操作数，也有个别指令有 3 个甚至 4 个操作数。多数操作数需要显式指明，有些操作数隐含使用。

8086 机器代码的一般格式如图 1-9 所示。操作码占 1 或 2 字节，后面的各字节指明操作数。其中，“mod reg r/m”字段表明寻找操作数的方法（即寻址方式，详见第 2 章），“位移量”字段给出相对基地址的偏移量，“立即数”字段给出操作数本身。

下面通过程序中使用最多的，也是指令系统中最基本的数据传送指令为例简单加以说明。

数据传送指令的助记符是“MOV”（取自 Move），功能是将数据从一个位置传送到另一个位置，类似高级语言的赋值语句。可以如下表达：

```
mov dest,src          ;dest ← src
```


src 表示要被传送的数据或数据所在的位置，称为源操作数（Source），书写在逗号之后。DEST 表示数据将要传送到的位置，称为目的操作数（Destination），书写在逗号之前。注意，后面分号是汇编语言使用的注释符号，表示其后内容是注释，可用于说明指令功能。



图 1-9 8086 机器代码的一般格式

例如，将 16 位寄存器 BX 的数据传送到 AX 寄存器的指令，可以书写为：

```
mov ax,bx ; 指令功能：AX←BX，指令代码：89 D8（十六进制表示）
```

其中第 1 个字节“89”是操作码，表示传送 16 位数据的 MOV 指令。第 2 个字节“D8”表示源操作数 BX，目的操作数 AX。

再如，将寄存器 BX 内容加寄存器 SI 内容，再加 6 的值作为存储器地址，从该地址单元传送一个字节数据给 AL 寄存器，可以书写为：

```
mov al,[bx+si+6] ; 指令功能：AL←[BX+SI+6]  
; 机器代码：8A 40 06（十六进制表示）
```

其中第 1 个操作码字节“8A”，表示传送 8 位数据的 MOV 指令。第 2 个字节“40”表示目的操作数是 AL、源操作数是通过寄存器 BX 和 SI 以及 8 位位移量相加得到存储器偏移地址，而第 3 个字节“06”正是这个位移量。

1.3.2 语句格式

像其他程序设计语言一样，汇编语言对其语句格式、程序结构以及开发过程等有相应的要求，它们本质上相同、方法上相似、具体内容各有特色。

汇编语言程序由语句序列构成，每条语句一般占一行，每行不超过 132 个字符（MASM 6.0 开始可以是 512 个字符）。汇编语言的语句一般都由分隔符分成的 4 个部分组成，有相似的两种格式，对应表达处理器指令和汇编程序伪指令。

（1）执行性语句——表达处理器指令的语句：

```
标号: 处理器指令助记符 操作数,操作数 ;注释
```

（2）说明性语句——表达汇编程序命令的语句：

```
名字 伪指令助记符 参数,参数,…… ;注释
```

一条执行性语句通常对应一条处理器指令，一般书写在代码段，是程序设计的主体部分。而说明性语句指示源程序如何汇编、变量怎样定义、过程怎么设置等。相对于真正的处理器指令（也称为真指令、硬指令），汇编程序命令也称为伪指令（Pseudoinstruction）、指示符（Directive）。

1. 标号与名字

执行性语句中，冒号前的标号表示处理器指令在主存中的逻辑地址，主要用于指示分支、循环等程序的目的地址，可有可无。说明性语句中的名字可以是变量名、段名、子程序名等，反映变量、段和子程序等的逻辑地址。标号采用冒号分隔处理器指令，名字采用空格或制表符分隔伪指令，据此也分别了两种语句。

标号和名字是符合汇编程序语法的用户自定义的标识符 (Identifier)。标识符 (也称为符号 Symbol) 一般最多由 31 个字母、数字及规定的特殊符号 (如 `_`、`$`、`?`、`@`) 组成, 不能以数字开头 (与高级程序语言一样)。在一个源程序中, 用户定义的每个标识符必须是唯一的, 且不能是汇编程序采用的保留字。保留字 (Reserved Word) 是编程语言本身需要使用的各种具有特定含义的标识符, 也称为关键字 (Key Word)。汇编程序中主要有处理器指令助记符、伪指令助记符、操作符、寄存器名以及预定义符号等。

例如, `msg`、`var2`、`buf`、`next`、`again` 都是合法的用户自定义标识符。而 `8var`、`ax`、`mov`、`db` 则是不符合语法 (非法) 的标识符, 原因是: `8var` 以数字开头, 其他是保留字。`ax` 是寄存器名、`mov` 是指令助记符、`db` 是伪指令助记符。

默认情况下, 汇编程序不区分包括保留字在内的标识符字母大小写。换句话说, 汇编语言是大小写不敏感的。例如, 对于寄存器名 `AX`, 还可以书写成 `ax`、`Ax` 等。`msg` 变量名, 还可以 `Msg`、`MSG` 等形式出现, 它们表达同一个变量。本书处理的原则是: 程序中一般使用小写字母形式, 功能注释、文字说明通常采用大写字母形式。

用户自定义标识符时, 应尽量具有描述性并易于理解, 一般不建议使用特殊符号开头, 因为特殊符号没有含义, 而且常被编译 (汇编) 程序所使用, 例如 C 语言编译程序在内部为函数增加 “`_`” 前缀, `MASM` 大量使用 “`@`” 作为预定义符号的前缀。如果你不确信标识符可用, 就不要使用它。一个简单的规则是: 以字母开头, 后跟字母或数字。

2. 助记符

助记符 (Mnemonics) 是帮助记忆指令的符号, 反映指令的功能。

处理器指令助记符可以是任何一条处理器指令, 表示一种处理器操作。同一系列处理器的指令常会增加, 不同系列处理器的指令系统不尽相同。例如, 前面介绍的数据传送指令, 其助记符是 “`MOV`”, 调用中断服务程序的指令助记符是 “`INT`” (Interrupt), 调用子程序的指令助记符是 “`CALL`”。

将数字 9 传送到 `AH` 寄存器的汇编语言执行性语句书写为:

```
mov ah,9           ;使得 AH=9
```

伪指令助记符由汇编程序定义, 表达一个汇编过程中的命令, 随着汇编程序版本增加, 伪指令会增加、功能也会增强。例如, 汇编语言程序中频繁使用的字节变量定义伪指令, 其助记符是 “`DB`” (Define Byte), 功能是在主存中分配若干的存储空间, 用于保存变量值, 该变量以字节为单位存取。

用 `DB` 伪指令定义一个字符串, 并取名字 `MSG` 指示, 这个汇编语言说明性语句书写为:

```
msg                db 'Hello, Assembly!',13,10,'$'
```

名字 `MSG` 指示这个字符串在主存的逻辑地址, 包含有段基地址和偏移地址, 也就是这个字符串的变量名。可以用一个 `MASM` 操作符 `OFFSET` 获得其偏移地址, 保存到 `DX` 寄存器, 汇编语言执行性语句如下:

```
mov dx,offset msg   ;DX 获得 MSG 的偏移地址
```

`MASM` 操作符 (Operator) 是对常量、变量、地址等进行操作的关键字。例如, 进行加减乘除运算的操作符 (也称运算符) 与高级程序语言一样, 依次是英文符号: `+`、`-`、`*` 和 `/`。

3. 操作数和参数

处理器指令的操作数表示参与操作的对象, 可以是一个具体的常量、也可以是保存在寄

存器的数据，还可以是一个保存在存储器中的变量等。双操作数的指令中，目的操作数写在逗号前，还可用来存放指令操作的结果；对应地，逗号后的操作数就称为源操作数。

例如，指令“MOV AH,9”中数字9是常量形式的源操作数，AH是寄存器形式的目的操作数。同样，OFFSET MSG经汇编后转换为一个具体的偏移地址，也是常量。

伪指令的参数可以是常量、变量名、表达式等，可以有多个，参数之间用逗号分隔。例如在“Hello, Assembly !,13,10,\$”示例中，就用单引号表达了一个字符串“Hello, Assembly !”，一个字符“\$”，还有常量13和10（这两个常量在ASCII码表中表示回车和换行控制字符，其作用相当于C语言的“\n”）。

4. 注释和分隔符

汇编语言语句中，分号后的内容是注释，它通常是对指令或程序片段功能的说明，是为了程序便于阅读而加上的，不是必须有的。必要时，一个语句行也可以由分号开始作为阶段性注释。汇编程序在翻译源程序时将跳过该部分，不对它们做任何处理。

建议大家一定要养成书写注释的良好习惯。

汇编语言语句的4个组成部分要用分隔符分开。标号后的冒号、注释前的分号以及操作数间和参数间的逗号都是规定采用的分隔符，其他部分通常采用空格或制表符作为分隔符。多个空格和制表符的作用与一个相同。另外，MASM也支持续行符“\”，表示本行内容与上一行内容属于同一个语句。注释可以使用英文书写。在支持汉字的编辑环境当然也可以使用汉字进行程序注释，但注意这些分隔符都必须使用英文标点，否则无法通过汇编。

良好的语句格式有利于编程，尤其是源程序阅读。在本书的汇编语言源程序中，标号和名字从首列开始书写；通过制表符对齐各个语句行的助记符；助记符之后用空格分隔操作数和参数部分（多个操作数和参数，按照语法要求使用逗号分隔）；再利用制表符对齐注释部分。

1.3.3 源程序框架

对应存储空间的分段管理，用汇编语言编程时也常将源程序分成代码段、数据段或堆栈段。需要独立运行的程序必须包含一个代码段，并指示程序执行的起始位置。需要执行的可执行性语句必须位于某一个代码段内。说明性语句通常安排在数据段，或根据需要位于其他段。

随着MS-DOS和MASM的发展，MASM各版本支持多种汇编语言源程序格式，本书介绍一个最简单的程序格式，使用MASM 6.x版本的简化段定义源程序格式，程序模板如下：

example.asm in DOS

```
.model small    ;定义程序的存储模型 (small 表示小型模型)
.stack          ;定义堆栈段 (默认是 1KB 空间)
.data           ;定义数据段
.....         ;数据定义 (数据待填)
.code           ;定义代码段
.startup        ;程序执行起始，同时设置数据段寄存器 DS 指向程序的数据段
.....         ;主程序 (指令待填)
.exit           ;程序执行结束，返回 DOS
.....         ;子程序 (指令待填)
end             ;汇编结束
```

在简化段定义 (Simplified Segment Definition) 的源程序格式中，以圆点 (即英文小数点、

句号) 开始的伪指令说明程序的结构, 必须具有存储模式伪指令.MODEL。随后.STACK, .DATA 和.CODE 依次定义堆栈段、数据段和代码段, 一个段的开始自动结束上一个段。代码段中, 通过.STARTUP 语句说明程序从该处开始执行, 并含有给 DS 赋值、使其指向该程序的数据段功能, 便于后续指令访问数据段中的数据。程序最后利用.EXIT 指令说明本程序执行结束、返回 DOS 操作系统。

1. 程序的存储模型

存储模型 (Memory Model) 决定一个程序的规模, 也确定进行子程序调用、指令转移和数据访问的默认属性。当使用简化段定义的源程序格式时, 必须有存储模型.MODEL 语句, 且位于所有简化段定义语句之前。其格式为:

.model 存储模型, 语言类型

.MODEL 语句确定了程序采用的存储模型, MASM 有 7 种可以选择, 如表 1-5 所示。

表 1-5 存储模型

存储模型	特 点
TINY (微型模型)	创建 COM 类型程序, 只有一个小于 64KB 的逻辑段 (MASM 6.x 支持)
SMALL (小型模型)	创建小应用程序, 只有一个代码段和一个数据段, 每个段不大于 64KB
COMPACT (紧凑模型)	创建代码少、数据多的程序, 只有一个代码段 (不大于 64KB), 但可有多个数据段
MEDIUM (中型模型)	创建代码多、数据少的程序, 可有多个代码段, 但只有一个数据段 (不大于 64KB)
LARGE (大型模型)	创建大应用程序, 可有多个代码段和多个数据段 (静态数据小于 64KB)
HUGE (巨型模型)	创建更大的应用程序, 可有多个代码段和数据段, 对静态数据没有限制
FLAT (平展模型)	创建一个 32 位的程序, 运行在 IA-32 微处理器的 32 位 Windows 操作系统

创建运行于 DOS 操作系统下的应用程序, 可根据需要选择前 6 种模型, 一般的小型程序 (例如学习中的小程序) 可以选用 SMALL 模型, 大型程序选择 LARGE 模型。要创建 COM 程序只能用 TINY 模型, 其他模型产生 EXE 程序。FLAT 模型只能用于 32 位 Windows 应用程序中, 不能在 DOS 环境执行。

DOS 环境的 COM 类型程序要求将程序的代码、数据和堆栈都安排在一个逻辑段中, 大小不超过 64KB, 是一种比较紧凑的程序格式。在上述程序模板中只要将.STACK 和.DATA 语句去掉, 并将数据定义填到子程序之后 (END 之前) 就形成了一个 COM 类型程序的模板文件。

语言类型主要用于与其他语言混合编程时约定调用的规范, 例如 Windows 的系统函数采用标准调用语言类型 “STDCALL”。MASM 汇编程序还支持 C 语言调用规范, 其关键字是 “C”。编写 DOS 应用程序通常不需要。

2. 逻辑段的简化定义

堆栈段定义伪指令.STACK 创建一个堆栈段, 段名是: STACK。保留字后可书写一个数值型参数指定堆栈段所占存储空间的字节数, 默认是 1KB (即 1024B=400H 字节)。堆栈段名可用@STACK 预定义操作符表示。

数据段定义伪指令.DATA 创建一个数据段, 段名是: _DATA。数据段名可用@DATA 预定义操作符表示。

代码段定义伪指令.CODE 创建一个代码段, 后可选一个标识符型参数指定该代码段的段名。如果没有给出段名, 则采用默认段名, 例如在 TINY、SMALL、COMPACT 和 FLAT 模

式下，默认的代码段名是：_TEXT。代码段名可用@CODE 预定义操作符表示。

3. 程序执行的开始

MASM 6.0 引入的.STARTUP 指令指明了本程序开始执行的位置，并同时使数据段寄存器 DS 等于用.DATA 伪指令定义的数据段的段基地址。

MASM 汇编程序在对源程序的汇编、连接过程中，会根据程序起始位置正确地设置代码段的 CS 和 IP 值，根据堆栈大小设置堆栈段的 SS 和 SP 值，但并不设置 DS 和 ES 值。这样，程序如果使用数据段，就必须在代码段中明确给 DS 等寄存器赋值。由于大多数程序需要在数据段定义变量，所以通常应该赋值给 DS；使用到附加段就一定设置 ES。

如果不使用.STARTUP 指令，通常可以用如下两条语句代替：

```
start:  mov ax,@data    ;@DATA 表示数据段的段地址，先传送给 AX 寄存器
        mov ds,ax      ;设置 DS 等于 AX，即数据段的段地址
```

其中标号 start 也可以使用其他标识符，此处需要利用它在最后的汇编结束 END 指令作为参数，用于指明程序开始执行的位置。

4. 程序执行的终止

应用程序执行结束，应该将控制权交还给操作系统，.EXIT 指令就实现了此功能。它实际上是利用了 DOS 功能调用的 4CH 号功能实现的，所以可以用如下两条语句代替：

```
mov ah,4ch
int 21h
```

DOS 功能调用是 MS-DOS 操作系统提供给程序员的一些子程序库，主要以第 21H 号中断的形式使用（即指令“INT 21H”，参见下面介绍）。

5. 源程序的汇编结束

汇编结束表示汇编程序到此结束将源程序翻译成目标模块代码的过程，它不是指程序终止执行。源程序的最后必须有一条 END 伪指令，END 指令之后的任何内容将不会被汇编程序所理会。

END 伪指令后面可以有一个“标号”性质的参数，用于指定程序开始执行于该标号所指示的指令。汇编程序将据此设置 CS 和 IP 值。如果没有.STARTUP 指令说明程序开始执行的位置，就需要利用这种方法指明。例如，对应前面我们使用的 START 标号，这时就需要用如下语句代替原“END”语句：

```
end start
```

现在我们用上述程序格式，编写一个在屏幕上显示信息的程序。

【例 1-1】信息显示程序。

在数据段给出这个字符串形式的信息，采用字节定义伪指令 DB 实现：

```
        ;数据段
msg     db 'Hello, Assembly!',13,10,'$'      ;定义要显示的字符串
```

在代码段编写显示字符串的程序：

```
        ;代码段
mov dx,offset msg      ; (1) 指定字符串在数据段的偏移地址
mov ah,9               ; (2) AH 赋值 9
int 21h                ; (3) 利用功能调用显示信息
```

这里使用了 DOS 的 9 号功能实现字符串的显示（参见附录 B），它要求：

（1）设置入口参数：DS:DX=字符串在主存逻辑段中的段地址:偏移地址。同时还要求保存在主存的字符串以“\$”作为结尾符（类似 C/C++语言中隐含用 NULL 作为字符串结尾）。

（2）需要将 AH 赋值 9，即使用 9 号 DOS 功能。

（3）使用指令“INT 21H”实现 DOS 功能的调用。

由于将字符串安排在了数据段，而 STARTUP 指令已经将 DS 赋值为数据段基地址，所以程序中只要将字符串的偏移地址传送给 DX 就完成入口参数的设置，执行 DOS 功能调用就可以实现输出了。

将例题程序填入程序模板（EXAMPLE.ASM）预留的位置，即将数据段内容填入数据段定义指令.DATA 之后，将代码段内容填入程序开始执行 STARTUP 指令之后，就编制了一个完整的 MASM 汇编语言源程序。

```
                ;eg101.asm (文件名)
.model small
.stack
.data
msg             db 'Hello, Assembly !',13,10,'$'          ;定义要显示的字符串
.code
.startup
mov dx,offset msg                ; (1) 指定字符串在数据段的偏移地址
mov ah,9                    ; (2) AH 赋值 9
int 21h                    ; (3) 利用功能调用显示信息
.exit
end
```

需要说明的是，本书前 5 章例题程序都将如此处理，教材只给出数据段的变量定义、主程序和子程序代码等部分（除非有特别说明），以便将注意力集中于编程本身（而不是被烦琐的程序格式所困惑）。读者可以基于模板文件的源程序框架创建完整的源程序文件，也可以参考配套软件包中提供的所有例题程序的源程序文件。

还记得你编写的第一个 C 语言程序吗？那个经典的显示“Hello, World!”程序！现在，对绝大多数读者来说，都是从高级语言开始熟悉计算机程序设计的。虽然汇编语言不是高级语言，但它们都是程序设计语言，有许多本质上相同或相通的方面。所以，学习过程中不妨做些简单对比；这样，既可以巩固高级语言的知识，也有利于熟悉汇编语言，通过汇编语言还可以进一步加深对高级语言的理解。

6. DOS 功能调用

汇编语言怎么总是使用操作系统的功能调用呢？使用一种编程语言进行程序设计，程序员需要利用其开发环境提供的各种功能，例如函数、程序库。如果这些功能无法满足程序员的要求，还可以直接利用操作系统提供的程序库，否则只有自己编写特定的程序。汇编语言作为一种低级程序设计语言，汇编程序通常并没有为其提供任何函数或程序库，所以必须利用操作系统的编程资源。显然，这是进行程序设计，尤其是采用汇编语言进行程序设计必须掌握的一个重要方面。DOS 提供给程序员的编程资源是以中断调用方法使用的各种子程序，Windows 则以应用程序接口 API 形式提供动态连接库 DLL。

中断是一种增强处理器功能的机制，中断调用是借助中断机制改变程序执行顺序的方

法，类似于汇编语言的子程序调用（对应高级语言的函数调用）。8086 处理器支持 256 个中断，每个中断用一个中断编号区别，即中断 0~中断 255。中断调用指令“INT N”实现调用 N 号中断服务程序的功能。DOS 系统中，主要分配 21H 号中断用于程序员调用 DOS 操作系统功能。

调用 DOS 操作系统功能的一般方法如下：

- (1) 在 AH 寄存器中设置系统功能调用号，说明选择的功能。
- (2) 在指定寄存器中设置入口参数，以便按照要求执行功能。
- (3) 用中断调用指令“INT 21H”执行功能调用。
- (4) 根据出口参数分析功能调用执行情况。

实际上，这就是调用子程序的一般步骤（类似高级语言调用函数）。根据功能不同，有些没有入口参数或出口参数，有些入口参数或出口参数可能较复杂，并且可能有一些特殊要求。

表 1-6 罗列了本书主要使用的 DOS 基本功能调用，更详细的说明参见附录 B。

表 1-6 DOS 基本功能调用（INT 21H）

子功能号	功 能	入口参数	出口参数
AH=01H	从标准输入设备输入一个字符		AL=输入字符的 ASCII 码
AH=02H	向标准输出设备输出一个字符	DL=字符的 ASCII 码	
AH=09H	向标准输出设备输出一个字符串	DX=字符串地址（以\$结尾）	
AH=4CH	程序执行终止	AL=返回代码	

例如 4CH 号功能是返回 DOS，它可以有一个入口参数，就是使 AL 等于程序的返回代码（通常用 0 表示程序没有错误）。这样，正常返回 DOS 可以增加一条指令“MOV AL,0”，或者用“MOV AX,4C00H”代替“MOV AH,4CH”指令。这时，它就相当于“.EXIT 0”语句。

7. 输入输出子程序库

程序运行需要与用户进行交互，但操作系统往往只提供对字符（串）的输入输出，不能直接实现诸如十进制、十六进制等数据形式的输入输出。为了方便进行键盘输入和显示器输出，本书作者精心编制了一个输入输出子程序库 IO.LIB（详见附录 C），配合有一个声明文件 IO.INC。汇编语言程序员只要利用源文件包含伪指令 INCLUDE 声明，并将这两个文件复制在源程序文件所在的目录下，就可以使用子程序调用指令 CALL 调用其中的子程序，执行其功能。

在本书提供的子程序库中，子程序名以 READ 开头表示键盘输入，以 DISP 开头表示显示器输出，调用的一般格式如下：

```
mov ax,入口参数
call 子程序名
```

例如，在当前光标位置显示字符串的功能是 DISPMSG 子程序。使用这个子程序，需要定义以 0 结尾的字符串，调用前赋值 AX 为该字符串的偏移地址，使用 CALL 指令实现调用。使用子程序库实现例 1-1 功能的汇编语言源程序如下：

```
;eg101a.asm
include io.inc
.model small
.stack
```

```

msg      .data
         db 'Hello, Assembly !',13,10,0      ;定义要显示的字符串
         .code
         .startup
         mov ax,offset msg                    ;指定字符串在数据段的偏移地址
         call dispmsg                         ;利用子程序库的子程序显示信息
         .exit
         end

```

注意，子程序 DISPMSG 要求定义的字符串以 0 结尾（C 语言格式），更具通用性；而 DOS 功能 9 号调用要求以字符“\$”结尾，无法显示“\$”本身。

为了便于应用本书配套的各种实用输入输出子程序，不妨总是在源程序开始加上“INCLUDE IO.INC”语句（使用程序模板 EXAMPLEA.ASM）。

1.4 汇编语言程序的开发

有了汇编语言程序，接着要生成可执行文件，查看运行结果。

1.4.1 开发环境

开发汇编语言程序，首先需要安装开发软件，熟悉开发平台。

1. 安装开发软件包

支持 Intel 80x86 处理器的汇编程序有很多。在 DOS 和 Windows 操作系统下，最流行微软汇编程序 MASM，Borland 公司的 TASM 也常用，两者相差不大。在 Linux 操作系统下，标准的汇编程序是 GAS，NASM 也较常用。

20 世纪 80 年代初微软公司推出 MASM 1.0。MASM 4.0 支持 80286/80287 的处理器和协处理器；MASM 5.0 支持 80386/80387 处理器和协处理器，并加进了简化段定义伪指令和存储模型伪指令，汇编和连接的速度更快。MASM 6.0 是 1991 年推出的，支持 80486 处理器，它对 MASM 进行重新组织，并提供了许多类似高级语言的新特点。MASM 6.0 之后又有一些改进，微软推出 MASM 6.11；利用它的免费补丁程序可以升级到 MASM 6.14，支持到 Pentium III 的多媒体指令系统。MASM 6.11 是最后一个独立发行的 MASM 软件包，这以后的 MASM 都存在于 Visual C++ 开发工具中，例如本书使用 Visual C++ 6.0 的 MASM 升级包中的 MASM 6.15，以便支持 Pentium 4 的 SSE2 指令系统。Visual C++.NET 2003 中有 MASM 7.10，但没有大的更新。Visual C++.NET 2005 提供的 MASM 8.0 才支持 Pentium 4 的 SSE3 指令系统，同时还提供了一个 ML64.EXE 程序用于支持 64 位指令系统。

安装 MASM 6.x 完全版，需要在 DOS（或 Windows 的 MS-DOS 模拟环境）下，运行其 SETUP.EXE 程序实现，通常选择在 MS-DOS / Microsoft Windows 操作系统下使用。

为便于初学者入门，本书整合 MASM 6.15 等相关文件，创建了一个软件包。它是一个基于 DOS 模拟环境（和 32 位控制台），虽基本但完整的 MASM 6.15 汇编语言开发系统（可从本书前言提供的网站下载，或者按照下面说明自行组建）。

建议将本书配套的 ML615 软件包安装到硬盘 D 分区的 ML615 目录（否则最好将 DOS.BAT 和 WIN.BAT 文件中，路径设置命令的“D:\ML615”相应修改为安装所在的分区和

目录)。本书的汇编语言程序开发基于本软件包，它主要包含如下内容：

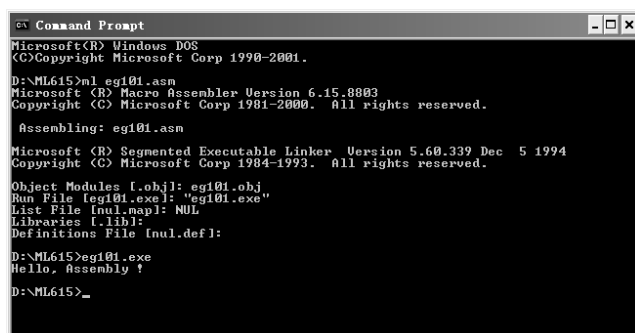
- ④ MASM 6.15 版本的汇编程序 ML.EXE 和配套的汇编错误信息文件 ML.ERR（取自微软 Visual C++ 6.0 配套的汇编语言升级包），连接程序 LINK.EXE 和子程序库管理文件 LIB.EXE（取自 MASM 6.11）；
- ④ CV 目录下的 CodeView 调试程序 CV.EXE 以及配套的库文件（取自 MASM 6.11）；
- ④ HELP 目录下的快速帮助文件 QH.EXE，以及 MASM 宏汇编语言、汇编程序 ML、连接程序 LINK、调试程序 CV 等所有帮助文件（取自 MASM 6.11）；
- ④ BIN32 目录下的开发 32 位控制台汇编语言程序需要的连接程序 LINK.EXE（与 DOS 环境的连接程序不同）、导入库文件 KERNEL32.LIB 和 USER32.LIB 等（取自 Visual C++ 6.0）；
- ④ PROGS 目录下的全书的例题程序文件；
- ④ 本书作者编写的 MS-DOS 环境的输入输出子程序库文件 IO.LIB 和配套的包含文件 IO.INC，以及简化输入的批处理文件。

例如，作者编辑的批处理文件 DOS.BAT 可以实现快速进入模拟 DOS 环境的当前目录。读者只要在 Windows 资源管理器中打开 MASM 所在的文件夹（建议是 D:\ML615），双击该批处理文件，就可以一步打开模拟 DOS 窗口，进入 MASM 开发环境。

进入模拟 DOS 的 MASM 目录后，输入一个简单的命令就可以生成可执行文件：

```
ml eg101.asm
```

该命令表示使用汇编程序 ML.EXE 对源程序文件 EG101.ASM 进行汇编和连接。如果没有错误，便可以运行程序，参见上述步骤的屏幕截图 1-10 如下所示。



```
Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.

D:\ML615>ml eg101.asm
Microsoft(R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.

Assembling: eg101.asm

Microsoft(R) Segmented Executable Linker Version 5.60.339 Dec 5 1994
Copyright (C) Microsoft Corp 1984-1993. All rights reserved.

Object Modules [obj]: eg101.obj
Run File [eg101.exe]: "eg101.exe"
List File [nul.map]: NUL
Libraries [lib]:
Definitions File [nul.def]:

D:\ML615>eg101.exe
Hello, Assembly!

D:\ML615>_
```

图 1-10 汇编连接并运行程序的过程

2. 进入模拟 DOS 环境

MASM 以 MS-DOS 操作系统为平台。DOS（Disk Operating System）虽然比较简单，但允许程序员访问任意资源，便于实践和实现，符合本课程的教学要求。读者可以使用 MS-DOS 启动机器（例如其最终版本 MS-DOS 6.22），但建议使用 Windows 的模拟 MS-DOS 环境。模拟 DOS 环境虽不是真正的 DOS 平台，但兼容绝大多数 DOS 应用程序，不但可以完全满足我们的教学需求（本书主体内容的前 6 章基于 DOS 模拟环境学习和实践汇编语言程序设计），还可以借助 Windows 的强大功能和良好保护。

在 Windows 操作系统的图形界面下，要进入模拟 DOS 环境，通常的操作方法是：

在屏幕左下角“开始—运行”打开的对话框中，输入“command”命令。

注意，模拟 DOS 环境执行的是 Windows 所在文件夹的 SYSTEM32 子文件夹下的

COMMAND.COM 文件。打开的窗口标题标示有“Command Prompt”或包含有“COMMAND.COM”。所以，为了避免与其他同名文件混淆，建立 DOS 模拟环境时可以输入完整的路径和命令“%SystemRoot%\system32\command.com”。其中“%SystemRoot%”表示 Windows 操作系统所在文件夹（例如，Windows XP 为 WINDOWS，Windows 2000 为 WINNT）。

通常，人们习惯用鼠标点击、逐步展开“开始—程序—附件—命令提示符”，或者在“开始—运行”对话框输入“cmd”命令打开一个酷似 DOS 的命令行窗口，但实际上它是 32 位 Windows 的控制台窗口，执行的是 Windows 所在文件夹的 SYSTEM32 子文件夹下的 CMD.EXE 文件。虽然 32 位控制台和模拟 DOS 环境的基本功能、操作和界面一致，但执行的文件不同，其实质不同。相对来说，CMD.EXE 命令行支持汉字输入输出、功能更强，打开的窗口标题标示有“命令提示符”或包含有“CMD.EXE”。

特别提醒注意，本书主要的应用程序基于 MS-DOS 模拟环境（COMMAND.COM），建议不要在 32 位 Windows 控制台环境（CMD.EXE）下运行，虽然很多时候也是正确的。这是因为，利用 DOS 功能调用编写的程序虽然可以在 32 位控制台环境执行，但不保证一定正确；同样，使用 32 位控制台 API 函数编写的程序也不保证一定在模拟 DOS 环境执行正确。

3. 进入 MASM 开发目录

进入模拟 DOS（或者 32 位控制台）后，还需要将 MASM 开发目录作为当前目录。

操作系统以目录（Directory）形式管理磁盘上的文件（Windows 为了使普通用户容易理解，使用了文件夹这个通俗的说法表示专业术语目录）。当我们指明某个文件时，为了区别于同名的其他文件，有必要说明该文件所在分区、根目录、各级子目录。上述分区和目录就是该文件的路径（Path），DOS 中利用向右的斜线“\”分隔各级目录。例如，在硬盘 D 分区根目录 ML615 的 PROGS 子目录的文件 EG101.ASM，需要如下表示：

```
d:\ml615\progs\eg101.asm
```

文件的完整路径称为“绝对路径”。采用这种指明文件的方法保证了唯一性，但未免有些烦琐。所以，我们还经常使用“相对路径”指明文件。采用相对路径首先必须明确相对位置，即当前所在的目录，简称当前目录（Current Directory）。实际上，在闪烁的 DOS 提示符前的路径就是当前目录所在位置。假如，D 分区当前目录是根目录 ML615，则指明上述 EG101.ASM 文件可以如下表达：

```
progs\eg101.asm
```

再如，PROGS 为当前目录，指明 ML615 目录下 ML.EXE 文件需要如下表达：

```
..\ml.exe
```

这里的两个小数点“..”表示当前目录的上级目录。另外，还经常使用“\”表示当前分区的根目录，用一个小数点“.”表示当前目录。

那么 DOS 下如何改变当前目录呢？这就要用到 DOS 内部命令 CD（Change Directory）。例如，进入模拟 DOS 后，可以首先键入分区字母加一个冒号从而进入需要的当前磁盘分区。然后键入 CD 命令，并用空格隔开需要进入的当前目录：

```
d:
cd \ml615
```

为了操作方便，可以定制一个进入 MASM 目录（假设在 D:\ML615）的 MS-DOS 快捷方式。以 Windows XP 操作系统为例，只要新建一个快捷方式，让其执行 COMMAND.COM 文件，并展开其属性中程序对话框，将“工作目录”文本框改为“D:\ML615”。这样双击这个

快捷方式就直接进入 MS-DOS 环境的“D:\ML615”目录。

在本书配套的软件包中，有一个批处理文件 DOS.BAT，在资源管理器下双击也能启动模拟 DOS 环境并快速进入 MASM 目录。批处理文件 DOS.BAT 的内容可以是：

```
@echo off
@set path=d:\ml615;%path%
%SystemRoot%\system32\command.com
@echo on
```

第 1 行命令表示不显示下面各行信息。第 2 行命令设置 D 分区 ML615 目录作为搜索路径，以便实际操作时能够执行这些目录下的文件，第 3 行执行操作系统所在根目录提供的 COMMAND.COM 进入模拟 DOS 环境窗口，并将 DOS.BAT 文件所在的目录（默认是 D:\ML615）作为当前目录。第 4 行命令表示以后输入的命令将显示出来。

4. 熟悉命令行操作

DOS 操作系统提供一个文本显示、字符输入的命令行操作界面。但实际上，命令行是一个基本的用户交互方式，图形用户界面只是其外壳（Shell）程序。Unix/Linux 等操作系统的基本用户操作界面也是命令行方式。如果不习惯命令行操作，建议通过运行 DOS（或 32 位控制台）内部或外部命令进行熟悉，这会有助于汇编语言程序的开发。

内部命令是 DOS（或 32 位控制台）本身具有的、直接支持的命令。进入 DOS 环境（或 32 位控制台）后只要键入其内部命令的关键字加上需要的参数就可以使用内部命令，例如改变目录（CD）、文件列表（DIR）、文件拷贝（COPY）、清除屏幕（CLS）、退出（EXIT）等常用命令。利用帮助命令（HELP）可以查看所有的内部命令和使用方法，也可以用命令加“/?”参数查询该命令的使用方法。

外部命令也是 DOS（或 32 位控制台）提供的命令，但它与其他可执行文件一样以文件形式保存在磁盘上，存放在 Windows 操作系统所在目录的 SYSTEM32 子目录下。由于操作系统通常已经将该目录列入为搜索路径，所以一般可以直接输入文件和参数执行外部命令。例如，使用 DOS 的调试程序 DEBUG.EXE，输入 DEBUG 就可以。

但是，对于没有建立搜索路径的其他可执行文件，或者存在多个同名的可执行文件时，执行这些文件需要先输入绝对路径或相对路径，然后再输入文件名，最后用空格分隔输入的参数。

如果没有指明路径，DOS（或 32 位控制台）将在当前目录下查找该文件；如果没有则在事先设置的搜索路径中依次查找；如果仍然没有查找到该文件，则将显示“'XX' 不是内部或外部命令，也不是可运行的程序或批处理文件”（'XX' is not recognized as an internal or external command, operable program or batch file.）。使用内部命令 PATH 可以查看和设置当前的搜索路径。所以，如果没有指明路径或者指明路径不正确，虽然文件存在但却会提示没有，或者执行了另外一个同名的文件。

DOS（和 32 位控制台）都支持扩展名为 EXE 的可执行文件，DOS 还支持扩展名为 COM 的可执行文件。DOS（和 32 位控制台）支持扩展名为 BAT 的批处理文件，它实际上是一个纯文本文件，其中编辑有依次执行的可执行文件名。如果执行外部命令时没有键入扩展名，则 DOS（或 32 位控制台）依次以 BAT、COM 和 EXE 为扩展名，先查找到哪个文件就执行哪个文件。

关闭 DOS 模拟窗口（或 32 位控制台）可以在命令行执行“EXIT”内部命令，或者使用鼠标单击“关闭”按钮。

1.4.2 开发过程

源程序的开发过程都需要编辑、编译（汇编）、连接等步骤，如图 1-11 所示。首先，用一个文本编辑器形成一个以 ASM 为扩展名的源程序文件；然后，用汇编程序翻译源程序，将 ASM 文件转换为 OBJ 目标模块文件；最后，用连接程序将一个或多个目标文件（含 LIB 库文件）连接成一个 EXE 可执行文件。

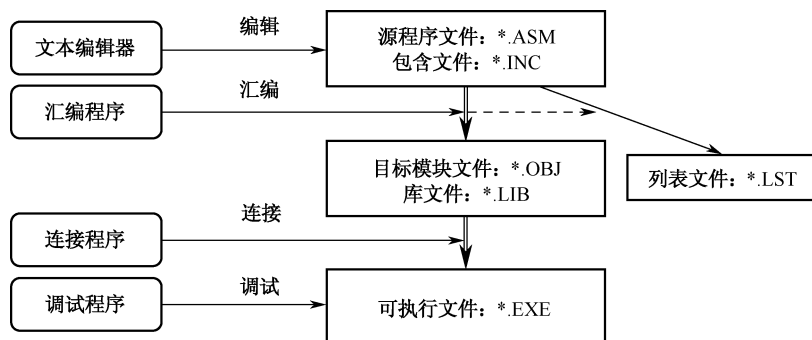


图 1-11 汇编语言程序的开发过程

1. 源程序的编辑

源程序文件的形成（编辑）可以通过任何一个文本编辑器实现，当然功能完善的编辑软件可提高编程效率。例如，大家可以使用 Windows 提供的记事本（Notepad），DOS 中的全屏文本编辑器 EDIT，甚至 MS Word。大家也可以使用已经熟悉的其他程序开发工具中的编辑环境，如：Visual C++ 或 Turbo C 的编辑器。一些专注于各种源程序文件编写的文本编辑软件也非常好用，值得推荐，例如 UltraEdit32（<http://www.ultraedit.com>）。

本书推荐使用 NOTEPAD2.EXE 程序。建议在其“设置”菜单中使用“文件关联”命令将汇编语言程序 ASM 文件与其建立关联（以后双击 ASM 程序就可以打开该记事本），还可以在“查看”菜单选择使用汇编程序语法高亮方案和语法高亮配置（便于区别助记符、数据等）。另外，要在“查看”菜单选中“行号”，这样该记事本可以给程序标示行号，以便出现错误时能够根据提示的行号快速定位到错误语句。

源程序文件是无格式文本文件，注意保存为纯文本类型，MASM 要求其源程序文件要以 ASM 为扩展名。现在将例 1-1 源程序输入编辑器，并以 EG101.ASM 为文件名保存在 ML615 目录下。为了便于操作，本书要求将源程序文件保存在 ML615 目录下，开发过程中生成的各种文件也自然存放于此，以避免指明文件路径的麻烦和出现找不到文件的错误。开发完成后的程序可以移动到另外一个目录下保存。为了便于管理，本书例题的源程序文件的命名规则是：EG 表示例题，前 1 个数字表示程序所在章号，后 2 个数字表示例题或习题序号，数字后的字母表示同一个程序的不同格式或解答。

2. 源程序的汇编

汇编是将汇编语言源程序翻译成由机器代码组成的目标模块文件的过程。MASM 6.x 提供的汇编程序是 ML.EXE。

进入已建立的 ML615 目录，输入如下命令及相应参数即可以完成源程序的汇编：

```
ml /c eg101.asm
```

其中，ML 表示运行 ML.EXE 程序，参数“/c”（小写字母，ML.EXE 的参数是大小写敏感的）表示仅利用 ML 实现源程序的汇编。参数也可以使用短线引导，例如：

```
ml -c eg101.asm
```

如果源程序中没有语法错误，MASM 将自动生成一个目标模块文件（EG101.OBJ），否则 MASM 将给出相应的错误信息。这时应根据错误信息，重新编辑修改源程序文件后，再进行汇编。

3. 目标文件的连接

连接程序能把一个或多个目标文件和库文件合成一个可执行文件。

在 ML615 目录下有了 EG101.OBJ 文件，输入如下命令实现目标文件的连接：

```
link eg101.obj
```

如果不带文件名，LINK 连接程序将提示输入 OBJ 文件名，它还会提示生成的可执行文件名以及列表文件名，一般（按回车键）采用默认文件名就可以。如果给出 EXE 文件名就可以替代与第一个模块文件名相同的默认名。给出 MAP 文件名将创建连接映象文件（它是一个文本文件，含有每个段在存储器中的分配情况），否则不生成映象文件。

如果没有严重错误，LINK 将生成一个可执行文件（EG101.EXE）；否则将提示相应的错误信息，这时需要根据错误信息重新修改源程序后再汇编、连接，直到生成可执行文件。

软件开发的主要步骤是编译（汇编）和连接。事实上，ML.EXE 汇编程序可以自动调用 LINK 连接程序（ML 表示 MASM 和 LINK），实现汇编和连接的依次进行。这只要在命令行中，输入不带“/c”参数的 ML 命令即可，例如：

```
ml eg101.asm
```

汇编程序 ML 和连接程序 LINK 支持很多参数，以便控制汇编和连接过程，用“/?”参数就可以看到帮助信息。例如，ML 可以用空格分隔多个 ASM 源程序文件，以便一次性汇编多个源文件。LINK 也可以将多个模块文件连接起来（用加号“+”分隔），形成一个可执行文件；还可以带 LIB 库文件进行连接。再如，ML 的参数“/FI”表示生成列表文件（下节介绍）；要在调试程序中直接使用程序定义的各种标识符，应该在 ML 命令增加参数“/ZI”，LINK 命令增加参数“/CO”，表示生成调试程序 Codeview 需要的符号信息。

4. 可执行程序的调试

生成可执行程序之后，就可以尝试运行。Windows 图形界面下，运行模拟 DOS 环境（或 32 位控制台）的可执行程序，需要首先进入模拟 DOS（或 32 位控制台）环境，然后在命令行提示符下输入文件名（可以省略扩展名），按下回车键：

```
eg101.exe
```

一般不要在 Windows 资源管理器下双击文件名启动 DOS（或控制台）可执行程序，这样往往看不到运行的显示结果，屏幕显示常一闪而过（程序已经执行，但打开的窗口随着执行结束马上被 Windows 操作系统关闭了）。

程序开发过程中难免会出现各种各样的错误。拼写有误的各种标识符（助记符、寄存器名、变量名等），形式写错的操作数和参数，不能支持的指令格式等都是常见的不符合汇编程序语法的错误，不过这些语法错误都会被汇编程序在汇编过程中指出，虽然有时并不准确。

连接过程中由于外部变量有错，其他模块文件、库文件不存在或者格式不符也会导致连接错误，不能生成可执行程序，连接程序也会指出错误原因。

一旦生成可执行文件，通常就意味着源程序没有语法错误和严重的连接错误，就可以尝试运行。如果没有看到期望的结果，往往预示着可能出现了逻辑错误或者运行错误。

排除程序存在的逻辑错误或运行错误，可以从源程序着手进行静态分析，即通过认真仔细阅读源程序代码，努力发现其中的错误，例如数据类型、数据结构是否正确或匹配，算法是否有不完善之处等。对于比较隐蔽难以发现，或者只有在运行的动态过程中才会出现的漏洞，就需要借助调试程序进行动态分析。

使用调试程序进行动态分析，有两种基本的调试方法。一种称为“断点调试”，在需要观察运行结果的语句位置设置断点（Breakpoint），程序执行到断点语句就暂停执行，等待用户分析。另一种称为“单步调试”，相当于每个语句都被调试程序设置了断点，每条语句执行后都暂停，用于细致跟踪程序运行轨迹。

学习过程中，也可以利用调试程序直观地查看指令的功能和程序执行过程。所以，在软件开发环境中，调试程序是不可或缺的一部分，有时还会用到反汇编程序等工具软件。

MS-DOS 提供 DEBUG.EXE 调试程序，本书前 5 章内容主要针对 8086 指令系统，可以使用 DEBUG 调试程序。但 DEBUG 功能简单，只支持 16 位 8086 处理器的整数指令和 8087 协处理器的浮点指令，不支持源程序级的调试。因此，MASM 提供了源代码调试程序 CodeView 4.10，可用于调试具有 32 位指令的 DOS 应用程序以及进行源程序级调试。

为了让调试程序方便进行源程序级调试，汇编时需要增加参数“/Zi”，连接命令增加参数“/CO”。为了方便操作，本书软件包中编辑有一个批处理文件 MAKE.BAT，已经将汇编和连接以及需要参数事先设置好，文件内容如下：

```
@echo off
rem make.bat, for assembling and linking 16-bit programs (.exe)
ml /c /FI /Sa /Zi %1.asm
if errorlevel 1 goto terminate
link /CO %1.obj;
if errorlevel 1 goto terminate
dir %1.*
:terminate
@echo on
```

REM 开头表示这是一个注释行。汇编和连接命令中使用“%1”代表输入的第一个文件名（扩展名已经表示出来，所以不需要输入英文句号及扩展名）。汇编和连接过程中没有错误，将在当前目录生成列表文件、目标文件和可执行文件等文件，并使用文件列表 DIR 命令进行了显示。如果汇编或连接有错误，“if-goto”命令将跳转到 terminate 位置，结束处理。

本书主要使用 DOS 操作系统自带的 DEBUG.EXE 调试程序，详细介绍参见附录 A，配合汇编语言的具体应用将在后续章节逐渐引入。

1.4.3 列表文件

源程序的汇编（编译）过程可以生成列表文件。

列表文件（List file）是一种文本文件，扩展名为 LST，含有源程序和目标代码，对学习汇编语言和发现错误很有用。创建列表文件，需要 ML 汇编程序使用“/FI”参数（大写字母

F, 接着小写字母 l, 不是数字 1), 例如输入如下命令:

```
ml /Fl eg101.asm
```

该命令除产生模块文件 EG101.OBJ 外, 还将生成列表文件 EG101.LST。列表文件有两部分内容, 第一部分是源程序及其代码, 如下所示:

```
eg101.asm                                Page 1 - 1
                                           ;eg101.asm
                                           .model small
                                           .stack
                                           .data
0000                                     .data
0000 48 65 6C 6C 6F 2C msg db 'Hello, Assembly !',13,10','$'
      20 41 73 73 65 6D
      62 6C 79 20 21 0D
      0A 24
0000                                     .code
                                           .startup
0017 BA 0000 R      mov dx,offset msg
001A B4 09          mov ah,9
001C CD 21          int 21h
                                           .exit
                                           end
```

列表文件第一部分中, 最左列是数据或指令在该段从 0 开始的相对偏移地址 (十六进制数形式), 接着相对偏移地址的是存放在主存的数据或指令的机器代码 (从低地址开始, 十六进制数形式, 操作码部分以字节为单位, 操作数则以数据类型为单位), 右边则是源程序文件中的汇编语言语句。机器代码后有字母 “R” 表示该指令代码的数值部分现在不能确定或只是相对地址, 它将在程序连接或进入主存时才能定位。调用指令代码后的字母 “E” 表示子程序来自外部 (External)。列表文件中常用的符号含义详见附录 D。

带有不同的参数进行汇编, 列表文件会有不同。例如, 再用空格分隔后加上 “/Sa” 参数, 会将汇编程序生成的代码也罗列出来, 这样.STARTUP 和.EXIT 语句所代表的汇编语言指令就可以观察到 (参见第 4 章)。

列表文件的第二部分是各种标识符的说明, 如下所示:

```
eg101.asm                                Symbols 2 - 1
Segments and Groups:
  Name                               Size      Length  Align  Combine  Class
DGROUP ..... GROUP
_DATA ..... 16 Bit      0014     Word   Public   'DATA'
STACK ..... 16 Bit      0400     Para   Stack   'STACK'
_TEXT ..... 16 Bit      0022     Word   Public   'CODE'

Symbols:
  Name                               Type      Value    Attr
@CodeSize ..... Number      0000h
@DataSize ..... Number      0000h
@Interface ..... Number      0000h
@Model ..... Number          0002h
@Startup ..... LNear         0000     _TEXT
```

```

@code ..... Text      _TEXT
@data ..... Text      DGROUP
@fardata? .....Text    FAR_BSS
@fardata ..... Text    FAR_DATA
@stack ..... Text      DGROUP
msg ..... Byte         0000  _DATA
0 Warnings
0 Errors

```

这部分列表文件中，罗列程序中使用的宏（Macros）、段和组（Segments and Groups）以及标号、变量名、子程序名等符号（Symbols）的有关信息。这些信息包括类型（Type）、段的操作数和地址长度（Size）、段的字节数量（Length）、变量的初始数值（Value）等。例如，代码段（段名_TEXT）属于 16 位（16 Bit）性质的逻辑段，共有 0022H（即十进制 34）个字节的指令代码，采用字（Word）对齐定位形式（Align），具有公用（Public）的组合类型（Combine）和‘CODE’类别（Class）。再如，字符串变量 msg 具有字节（Byte）类型，具有数值 0000（这里表示相对偏移地址），在数据段（_DATA）。这些内容需要学习第 2 章后才能理解。

列表文件最后总结了汇编过程中出现错误和警告的数量。错误（Error）是比较严重的语法错误，不能产生机器代码或产生的代码可能无法正确运行；警告（Warning）一般是不太关键的语法错误，有些警告也不影响程序的正确性。如果程序中有错误或警告，会在相应语句位置进行提示，并说明错误原因。

在图形界面的集成开发环境下，简单地单击创建（Build）按钮就可生成可执行文件。也许读者会感觉汇编语言的开发过程还挺复杂的，但实际上高级语言的开发过程也需要这些步骤，甚至更多步骤。例如：除源程序编辑、可执行文件调试外，在编译之前，C 语言程序还需要预处理，编译步骤也通常需要再分解为将源程序翻译成汇编语言代码或者中间代码（一般称“编译”），再将汇编语言代码（中间代码）翻译（一般称“汇编”）成目标代码的不同阶段，然后再连接生成可执行文件。而且，在编译步骤中也可以生成列表文件，其列表文件中有高级语言代码、目标代码，还可以有汇编语言代码。由于编译系统本身比较复杂，各种不同的编译系统生成的汇编语言代码格式也不尽相同，读者不可能（也没有必要）完全理解其列表文件，但仍然可以对比学习（参考本书第 8 章）。

因此，集成环境封装了高级语言整个开发过程，简单到一个鼠标动作；但是，学习汇编语言时，我们有必要了解底层内容，进而更深入地理解高层内容，更好地应用高级语言。所以，学习汇编语言需要读者沉下心来，认真体会每个细节，所谓“细节决定成败”，然也！

习 题 1

1.1 简答题

- (1) 传统计算机的 5 大部件演变为现代计算机的哪 3 个硬件子系统？
- (2) 汇编语言最主要的优势是什么？
- (3) 什么是通用寄存器？
- (4) 堆栈的存取原则是什么？
- (5) 标志寄存器主要保存哪方面的信息？
- (6) 最高有效位 MSB 是指哪一位？

- (7) 汇编语言中的标识符与高级语言的变量和常量名的组成原则有本质的区别吗?
- (8) 汇编语言的标识符大小写不敏感意味着什么?
- (9) 汇编语言源程序文件中, END 语句后的语句会被汇编吗?
- (10) 汇编时生成的列表文件主要包括哪些内容?

1.2 判断题

- (1) AX 被称为累加器, 在 8086 程序中使用很频繁。
- (2) 指令指针 IP 寄存器属于通用寄存器。
- (3) 8086 具有 8 个 32 位通用寄存器。
- (4) 8086 编程使用逻辑地址, 将其中段地址左移 4 位加偏移地址就是物理地址。
- (5) Windows 的模拟 DOS 环境与控制台环境是一样的。
- (6) 处理器的传送指令 MOV 属于汇编语言的执行性语句。
- (7) 汇编语言的语句由明显的 4 部分组成, 不需要分隔符区别。
- (8) MASM 汇编语言的注释用分号开始, 但不能用中文分号。
- (9) 程序终止执行也就意味着汇编结束, 所以两者含义相同。
- (10) 源程序文件和列表文件都是文本性质的文件。

1.3 填空题

- (1) 8086 处理器支持_____容量主存空间, 因为它有 20 个地址总线信号。
 - (2) 一个比特位是一个二进制位, _____位则被称为一个字节。
 - (3) 8086 处理器有 8 个 16 位通用寄存器, 其中 AX, _____, _____和 DX, 可以分成 8 位操作; 还有另外 4 个是_____, _____, _____和_____。
 - (4) 寄存器 DX 是_____位的, 但可以分成两个 8 位的寄存器, 其中 D0~D7 和 D8~D15 部分可以分别用名称_____和_____表示。
 - (5) 8086 处理器有_____个段寄存器, 它们都是_____位的。
 - (6) 8086 分段管理主存储器, 但要求段起始于_____的物理地址位置, 并且每段最大不超过_____。
 - (7) 逻辑地址由_____和_____两部分组成。代码段中下一条要执行的指令由 CS 和_____寄存器指示。
 - (8) Windows 的文件夹对应的专业术语是_____。
 - (9) 指令由表示指令功能的_____和表示操作对象的_____部分组成。
 - (10) MASM 要求汇编语言源程序文件的扩展名是_____, 汇编产生扩展名为 OBJ 的文件被称为_____文件, 可执行文件通常使用_____扩展名。
- 1.4 说明计算机系统的硬件组成及各部分作用。
- 1.5 什么是标志? 说出 8086 状态标志的名称和符号。
- 1.6 将如下 8086 的逻辑地址用其物理地址表示 (均为十六进制形式):
- (1) FFFF:0 (2) 40:17 (3) 2000:4500 (4) B821:4567
- 1.7 应用程序中主要有哪 3 类基本段, 各有什么用途?
- 1.8 说明高级语言、汇编语言、机器语言三者的区别, 谈谈你对汇编语言的认识。
- 1.9 区别如下概念: 助记符、汇编语言、汇编语言程序和汇编程序。
- 1.10 区别如下概念: 路径、绝对路径、相对路径、当前目录。系统磁盘上存在某个可执行文件, 但在 DOS 环境输入其文件名却提示没有这个文件, 是什么原因?

- 1.11 汇编语言语句有哪两种，每种语句由哪 4 个部分组成？
- 1.12 什么是标识符和保留字，汇编语言程序中标识符怎样组成？
- 1.13 MASM 汇编语言中，下面哪些是程序员可以使用的自定义标识符。
FFH, DS, Again, next, @data, h_ascii, 6364b, small
- 1.14 汇编语言程序的开发有哪 4 个步骤，分别利用什么程序完成、产生什么输出文件。

第 2 章 数据表示和寻址

数据（Data）是计算机处理的对象，处理器指令操作的对象也称操作数（Operand）。计算机中的数据需要使用二进制的 0 和 1 组合表示，程序设计语言中使用常量和变量形式来表示和定义，处理器指令则以寻址方式访问数据进行处理。

本章首先介绍计算机中数值和字符的编码方法，然后了解汇编语言如何使用常量表达数据，其次说明汇编语言如何使用变量保存数据，最后学习处理器指令如何寻址数据。本章围绕计算机内部数据的工作原理展开，通过阅读简单的程序及其列表文件直观理解，同时进一步熟悉汇编语言的开发过程。

2.1 数据表示

计算机只能识别 0 和 1 两个数码，进入计算机的任何信息都要转换成 0 和 1 数码。整数指令支持的基本数据类型是 8、16、32、64 位无符号整数和有符号整数，也支持字符、字符串和 BCD 码操作。本节主要介绍这些数据类型的数据表示。

2.1.1 数制

人有 10 个手指，所以习惯了十进制计数。计算机的硬件基础是数字电路，它处理具有低电平和高电平两种稳定状态的电平信号，所以使用了二进制。为了便于表达二进制数，人们又常用到十六进制数。

1. 二进制

计算机中为便于存储及物理实现，采用二进制表达数值。二进制数的特点为：逢 2 进 1，由 0 和 1 两个数码组成，基数为 2，各个位权以 2^k 表示。

二进制数： $a_n a_{n-1} \cdots a_1 a_0 . b_1 b_2 \cdots b_m =$

$$a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_1 \times 2^1 + a_0 \times 2^0 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_m \times 2^{-m}$$

其中 a_i, b_j 非 0 即 1。

二进制数的算术运算类似于十进制，只不过是逢 2 进 1、借 1 当 2，表 2-1 是二进制运算规则。图 2-1 采用 4 位二进制数，示例了二进制数的加减乘除运算，注意加减法会出现进位或借位，乘积和被除数是双倍长的数据，除法有商和余数两个部分。

表 2-1 二进制运算规则

加法运算	减法运算	乘法运算
$1+0=1$	$1-0=1$	$1 \times 0=0$
$1+1=0$ （进位 1）	$1-1=0$	$1 \times 1=1$
$0+0=0$	$0-0=0$	$0 \times 0=0$
$0+1=1$	$0-1=1$ （借位 1）	$0 \times 1=0$

$$1101+0011=0000 \text{ (进位1)}$$

$$\begin{array}{r} 1101 \\ + 0011 \\ \hline 10000 \end{array}$$

(a) 加法

$$1101-0011=1010$$

$$\begin{array}{r} 1101 \\ - 0011 \\ \hline 1010 \end{array}$$

(b) 减法

$$1101 \times 0011 = 00100111$$

$$\begin{array}{r} 1101 \\ \times 0011 \\ \hline 1101 \\ + 11010 \\ \hline 100111 \end{array}$$

(c) 乘法

$$01001001 \div 1101 = 0101 \text{ (余数 1000)}$$

$$\begin{array}{r} 101 \\ 1101 \overline{) 01001001} \\ \underline{- 1101} \\ 010101 \\ \underline{- 1101} \\ 1000 \end{array}$$

(d) 除法

图 2-1 二进制数的算术运算

2. 十六进制

由于二进制数书写较长、难以辨认，因此常用易于与之转换的十六进制数来描述二进制数。十六进制数的基数是 16，共有 16 个数码：0、1、2、3、4、5、6、7、8、9 和 A、B、C、D、E、F（也可以使用小写字母 a~f，依次表示十进制的 10~15），逢 16 进位，各个位的位权为 16^k 。

十六进制数： $a_n a_{n-1} \cdots a_1 a_0 . b_1 b_2 \cdots b_m =$

$$a_n \times 16^n + a_{n-1} \times 16^{n-1} + \cdots + a_1 \times 16^1 + a_0 \times 16^0 + b_1 \times 16^{-1} + b_2 \times 16^{-2} + \cdots + b_m \times 16^{-m}$$

其中 a_i, b_j 为 0~9 及 A~F 中的一个数码。

十六进制数的加减运算也类似十进制，但注意逢 16 进位 1，借 1 当 16。

例如：

$$23D9H + 94BEH = B987H, A59FH - 62B8H = 42E7H$$

这里的后缀字母 H（或小写 h）表示十六进制形式表达的数据。涉及计算机学科知识的文献中，常使用十六进制数表达地址、数据、指令代码等，所以应该熟悉十六进制数的加减运算。

3. 数制之间的转换

(1) 二进制数、十六进制数转换为十进制数需按权展开。

例如：

$$0011.1010B = 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 3.625$$

$$1.2H = 1 \times 16^0 + 2 \times 16^{-1} = 1.125$$

这里的后缀字母 B（或小写 b）表示二进制形式表达的数据。

(2) 十进制数的整数部分转换为二进制和十六进制数可用除法，把要转换的十进制数的整数部分不断除以二进制数和十六进制数的基数 2 或 16，并记下余数，直到商为 0 为止。由最后一个余数起逆向取各个余数，则为该十进制数整数部分转换成的二进制数和十六进制数。

图 2-2 演示了转换 126 的过程，结果是： $126 = 01111110B$ ， $126 = 7EH$ 。

(3) 十进制数的小数部分转换为二进制数和十六进制数则分别乘以各自的基数，记录整数部分，直到小数部分为 0 为止。

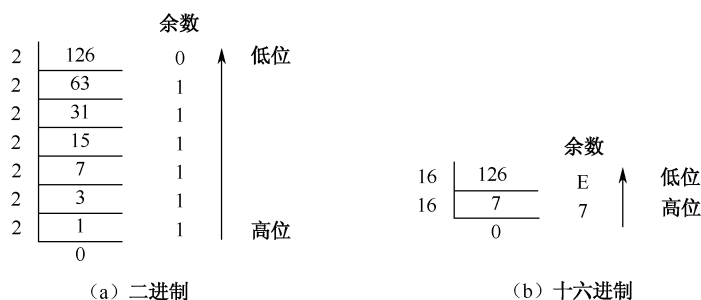


图 2-2 十进制整数的转换

图 2-3 演示了转换 0.8125 的过程，结果是：0.8125=0.1101B，0.8125=0.DH。

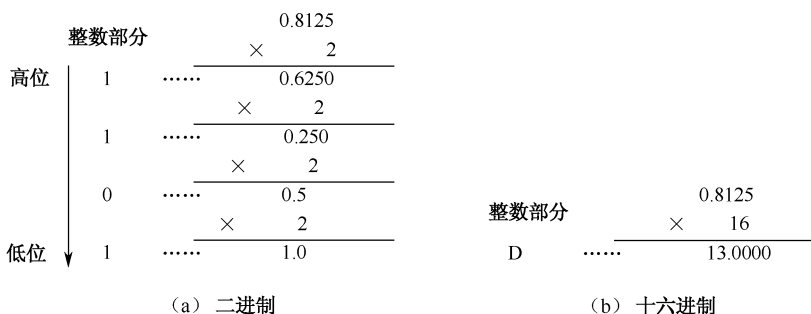


图 2-3 十进制小数的转换

小数部分的转换会发生总是无法乘到为 0 的情况，这时可选取一定位数（精度），当然这将产生无法避免的转换误差。

（4）二进制数和十六进制数之间具有对应关系：以小数点为基准，整数从右向左（从低位到高位）、小数从左向右（从高位到低位）每 4 个二进制位对应一个十六进制位，如表 2-2 所示，所以相互转换非常简单。表 2-2 还给出了 BCD 码以及常用的二进制位权值。

表 2-2 不同进制间（含 BCD 码）的对应关系

十 进 制	二 进 制	十六进制	BCD 码	常用二进制位权
0	0000	0	0	$2^{-3}=0.125$
1	0001	1	1	$2^{-2}=0.25$
2	0010	2	2	$2^{-1}=0.5$
3	0011	3	3	$2^0=1$
4	0100	4	4	$2^1=2$
5	0101	5	5	$2^2=4$
6	0110	6	6	$2^3=8$
7	0111	7	7	$2^4=16$
8	1000	8	8	$2^5=32$
9	1001	9	9	$2^6=64$
10	1010	A		$2^7=128$
11	1011	B		$2^8=256$
12	1100	C		$2^9=512$
13	1101	D		$2^{10}=1024$
14	1110	E		$2^{15}=32768$
15	1111	F		$2^{16}=65536$

例如：

00111010B=3AH, F2H=11110010B

2.1.2 数值的编码

编码是用文字、符号或者数码来表示某种信息（数值、语言、操作指令、状态等）的过程。组合 0 和 1 数码就是二进制编码。用 0 和 1 数码的组合在计算机中表达的数值被称为机器数；对应地，现实中真实的数值被称为真值。对数值来说，主要有两种编码方式：定点格式和浮点格式。定点整数是本书的主要讨论对象，浮点实数将在 9.1 节介绍。

1. 定点整数

定点格式固定小数点的位置表达数值，计算机中通常将数值表达成纯整数或纯小数，这种机器数称为定点数。整数可以将小数点固定在机器数的最右侧，实际上并不用表达出来，这就是整数处理器支持的定点整数，如图 2-4 所示。如果将小数点固定在机器数的最左侧就是定点小数。

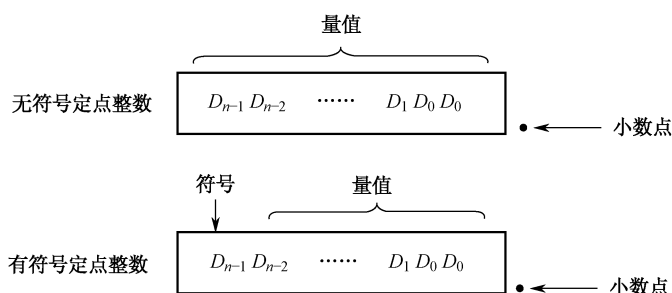


图 2-4 定点整数格式

定点整数如果不考虑正负，只表达 0 和正整数，就是无符号整数（简称无符号数）。在上面的数值转换和运算中，就默认采用无符号整数。8 位二进制有 256 个编码，依次是：00000000、00000001、00000010、……、11111110、11111111，使用十六进制形式是：00、01、02、……、FE、FF，对应表达无符号整数真值为：0、1、2、……、254、255。 N 位二进制共有 2^N 个编码，表达真值： $0 \sim 2^N - 1$ 。所以 16 位和 32 位二进制所能表示的无符号整数范围分别是： $0 \sim 2^{16} - 1$ 和 $0 \sim 2^{32} - 1$ 。

如果要表达数值正负，需要占用一个位，通常用机器数的最高位（故称为符号位），用 0 表示正数、1 表示负数，这就是有符号整数（简称有符号数或带符号数）。

2. 补码

有符号整数有多种表达形式，计算机中默认采用补码。因为采用补码，减法运算可以转换成加法运算，硬件电路只需设计加法器。

补码中最高位表示符号：正数用 0，负数用 1；正数补码同无符号数，直接表示数值大小；负数补码是将对应正数补码取反（即将 0 变为 1，1 变为 0），然后加 1 形成。

例如，正整数 105，用 8 位补码表示：

$[105]_{\text{补码}} = 01101001\text{B}$

负整数-105，用8位补码表示：

$$[-105]_{\text{补码}} = [01101001\text{B}]_{\text{取反}} + 1 = 10010110\text{B} + 1 = 10010111\text{B}$$

一个负数真值在用机器数补码表示时，需要一个“取反加1”的过程。同样，将一个最高位为1的补码（即真值为负数）转换成真值时，也需要一个“取反加1”的过程。

例如，补码：11100000B，真值： $-([11100000]_{\text{取反}} + 1) = -(00011111 + 1) = -00100000 = -2^5 = -32$ 。

进行负数求补运算，在数学上等效于用带借位的0作减法（下面等式中用中括号表达借位）。

例如：

$$\text{真值：}-8, \text{补码：} [-8]_{\text{补码}} = [1]0 - 8 = [1]00000000 - 00001000 = 11111000$$

$$\text{补码：} 11111000, \text{真值：} -([1]00000000 - 11111000) = -00001000 = -8$$

注意求补只针对负数进行，正数不需求补。另外，十六进制更便于表达，上述运算过程可以直接使用十六进制数。

由于符号要占用一个数位，8位二进制补码中只有7个数位表达数值，其所能表示的数值范围是： $-128 \sim -1, 0 \sim +127$ ，对应补码是：10000000~11111111、00000000~01111111，若用十六进制表达是：80~FF、00~7F。16位和32位二进制补码所能表示的数值范围分别是： $-2^{15} \sim +2^{15}-1$ 和 $-2^{31} \sim +2^{31}-1$ 。用N位二进制编码有符号整数，仍共有 2^N 个编码，但表达的真值范围是： $-2^{N-1} \sim +2^{N-1}-1$ 。

使用补码表达有符号整数，和无符号整数表达的数值个数一样，但范围不同。

2.1.3 字符的编码

在计算机中，各种字符需要用若干位的二进制码的组合表示，即字符的二进制编码。由于字节为计算机的基本存储单位，所以常用8个二进制位为单位表达字符。

1. BCD 码

一个十进制数位在计算机中用4位二进制编码来表示，这就是所谓的二进制编码的十进制数（Binary Coded Decimal，BCD）。常用的BCD码是8421 BCD码，它用4位二进制编码的低10个编码表示0~9这10个数字，参见表2-2。

BCD码很容易实现与十进制真值之间的转换。例如：

$$\text{BCD码：} 0100\ 1001\ 0111\ 1000.0001\ 0100\ 1001, \text{十进制真值：} \quad 4978.149$$

如果将二进制8位、即字节的高4位设置为0，仅用低4位表达一个BCD码，被称为非压缩（Unpacked）BCD码；而通常用1字节表达两位BCD码，就被称为压缩（Packed）BCD码。

BCD码虽然浪费了6个编码，但能够比较直观地表达十进制数，也容易与ASCII码相互转换、便于输入输出。另外它还可以比较精确地表达数据。例如，对于一个简单的数据0.2，采用浮点格式（详见9.1节）就无法精确表达。而采用BCD码便可以只使用4位“0010”来表达。最初的计算机支持十进制运算，8086处理器中使用调整指令实现十进制运算。

2. ASCII 码

字母和各种字符也必须按特定的规则用二进制编码才能在计算机中表示。编码方式可以有多种，其中最常用的一种编码是ASCII码（美国标准信息交换码：American Standard Code

for Information Interchange)。现在使用的 ASCII 码源于 20 世纪 50 年代，完成于 1967 年，由美国标准化组织 ANSI 定义在 ANSI X3.4-1986 中。

标准 ASCII 码用 7 位二进制编码，故有 128 个，如表 2-3 所示。计算机存储单位为 8 位，表达 ASCII 码时最高 D7 位通常为 0；通信时，D7 位通常用做奇偶校验位。

表 2-3 标准 ASCII 码及其字符

ASCII 码	字 符	ASCII 码	字 符	ASCII 码	字 符	ASCII 码	字 符
00H	NUL	20H	SP	40H	@	60H	`
01H	SOH	21H	!	41H	A	61H	a
02H	STX	22H	"	42H	B	62H	b
03H	ETX	23H	#	43H	C	63H	c
04H	EOT	24H	\$	44H	D	64H	d
05H	ENQ	25H	%	45H	E	65H	e
06H	ACK	26H	&	46H	F	66H	f
07H	BEL	27H	'	47H	G	67H	g
08H	BS	28H	(48H	H	68H	h
09H	HT	29H)	49H	I	69H	i
0AH	LF	2AH	*	4AH	J	6AH	j
0BH	VT	2BH	+	4BH	K	6BH	k
0CH	FF	2CH	,	4CH	L	6CH	l
0DH	CR	2DH	-	4DH	M	6DH	m
0EH	SO	2EH	.	4EH	N	6EH	n
0FH	SI	2FH	/	4FH	O	6FH	o
10H	DLE	30H	0	50H	P	70H	p
11H	DC1	31H	1	51H	Q	71H	q
12H	DC2	32H	2	52H	R	72H	r
13H	DC3	33H	3	53H	S	73H	s
14H	DC4	34H	4	54H	T	74H	t
15H	NAK	35H	5	55H	U	75H	u
16H	SYN	36H	6	56H	V	76H	v
17H	ETB	37H	7	57H	W	77H	w
18H	CAN	38H	8	58H	X	78H	x
19H	EM	39H	9	59H	Y	79H	y
1AH	SUB	3AH	:	5AH	Z	7AH	z
1BH	ESC	3BH	;	5BH	[7BH	{
1CH	FS	3CH	<	5CH	\	7CH	
1DH	GS	3DH	=	5DH]	7DH	}
1EH	RS	3EH	>	5EH	^	7EH	~
1FH	US	3FH	?	5FH	-	7FH	Del

ASCII 码表中的前 32 个和最后一个编码是不可显示的控制字符，用于表示某种操作。并不是所有设备都支持这些控制字符，也不是所有设备都按照同样的功能应用这些控制字符。不过，有些控制字符使用非常广泛，例如：0DH 表示回车 CR (Carriage Return)，控制屏幕光标时就是使光标回到本行首位；0AH 表示换行 LF (Line Feed)，就是使光标进入下一行，但列位置不变；08H 实现退格 BS (Backspace)，7FH 实现删除 DEL (Delete)。另外，07H

表示响铃 BEL (Bell), 1BH (ESC) 常对应键盘的 ESC 键 (多数人称其为 Escape 键)。ESC (Extra Services Control) 字符常与其他字符一起发送给外设 (例如打印机), 用于启动一种特殊功能, 很多程序中常使用它表示退出操作。

ASCII 码表中从 20H 开始 (含 20H) 的 95 个编码是可显示和打印的字符, 其中包括数码、英文字母、标点符号等。从表中可看到, 数码 0~9 的 ASCII 码为 30H~39H, 去掉高 4 位 (或者说减去 30H) 就是 BCD 码。大写字母 A~Z 的 ASCII 码为 41H~5AH, 而小写字母 a~z 则是 61H~7AH。大写字母和对应的小写字母相差 20H (32), 所以大小写字母很容易相互转换。ASCII 码中 20H 表示空格。尽管它显示空白, 但要占据一个字符的位置; 它也是一个字符, 表中用 SP (Space) 表示。熟悉这些字符的 ASCII 码规律对解决一些应用问题很有帮助, 例如英文字符就是按照其 ASCII 码大小进行排序的。

处理器只是按照二进制数操作字符编码, 并不区别可显示 (打印) 字符和非显示 (控制) 字符, 只有外部设备才区别对待, 产生不同的作用。例如, ASCII 字符设备总是以 ASCII 形式处理数据, 要显示 (打印) 数字 “8”, 必须将其 ASCII 码 (38H) 提供给显示器 (打印机)。

另外, PC 还采用扩展 ASCII 码, 主要表达各种制表用的符号等。扩展 ASCII 码最高 D7 位为 1, 以与标准 ASCII 码区别。

3. Unicode

ASCII 码表达了英文字符, 但却无法表达世界上所有语言的字符, 尤其是像非拉丁语系的语言, 例如中文、日文、韩文、阿拉伯文等。为此, 各国也都定义了各自的字符集, 但相互之间并不兼容。例如, 1981 年我国制定了《信息交换用汉字编码字符集基本集 GB2312-80》国家标准 (简称国标码)。规定每个汉字使用 16 位二进制编码, 即两个字节表达, 共计 7445 个汉字和字符。实际应用中, 为了保持与标准 ASCII 码兼容、不产生冲突, 国标码两个字节的最高位被设置为 1, 这称为汉字的机内码。不过, 汉字机内码会与扩展 ASCII 码冲突 (因它们的最高位都是 1), 所以一些西文制表符有时会显示为莫名其妙的汉字。

为了解决世界范围的信息交流问题, 1991 年国际上成立了统一码联盟 (Unicode Consortium), 制定了国际信息交换码 Unicode。在其网站上对 “什么是 Unicode” 给出了如下解答: “Unicode 给每个字符提供了一个唯一的数字, 不论是什么平台, 不论是什么程序, 不论是什么语言”。Unicode 使用 16 位编码, 能够对世界上所有语言的大多数字符进行编码, 并提供了扩展能力。Unicode 作为 ASCII 码的超集, 保持了与其兼容。Unicode 的前 256 个字符对应 ASCII 字符, 16 位编码的高字节为 0、低字节等于 ASCII 码值。例如, 大写字母 A 的 ASCII 码值是 41H, 用 Unicode 编码是 0041H。

现在 Unicode 已经越来越被大家认同, 很多程序设计语言和计算机系统都支持它。例如, Java 语言和微机 Windows 操作系统的默认字符集就是 Unicode。Unicode 标准还在发展, 2010 年 10 月 11 日完成 Unicode 6.0.0 版本, 详情访问统一码联盟网站 (<http://www.unicode.org>)。

2.2 常量表达

学习 C 语言时, 也是先讲解数据类型, 掌握如何使用常量和变量表达数据。C 语言的基本数据类型有字符 char、整型 int (包括短整型 short int 和长整型 long int), 以及浮点单精度 (float) 和双精度 (double) 实数。本章涉及最基本的整数编码, 包括整型和字符, 实际上字

符可以看做是 8 位的整数。两种浮点数据类型将在第 9 章介绍。

那么汇编语言如何使用常量和变量形式表达整数编码呢？基于前节基本知识，在本节先介绍如何使用常量表示数值和字符，下节说明怎样将它们保存在存储器中，最后再讲解处理器指令如何访问它们。

常量（Constant）是程序中使用的一个确定数值，在汇编语言中有多种表达形式。

1. 常数

这里的常数指由十、十六和二进制形式表达的数值，如表 2-4 所示。各种进制的数据以后缀字母区分，不加后缀字母的默认为十进制数。十六进制常数若以字母 A~F 开头，则要添加前导 0，以避免与不能以数字开头的标识符混淆。例如十进制数 10，用十六进制表达为 A，在汇编语言中需要表达成 0AH；如果不用前导 0，则与寄存器名 AH 相混淆。在 C 和 C++ 语言中，十六进制数使用 0x 前导，就不会出现这个问题。

表 2-4 各种进制的常数

进 制	数字组成	举 例
十进制	由 0~9 数字组成，以字母 D 或 d 结尾（默认，可以省略）	100, 255D
十六进制	由 0~9, A~F 组成，以字母 H 或 h 结尾； 以字母 A~F 开头前面要用 0 表达，以避免与标识符混淆	64H, 0FFH 0B800H
二进制	由 0 或 1 两个数字组成，以字母 B 或 b 结尾	01101100B

在实际应用中，通常使用十进制表达数值，使用十六进制表达存储器地址、BCD 码数值、数值的内部编码或者指令代码等，使用二进制表达需要进行位操作的数值（例如逻辑量）等。程序设计语言通常都支持八进制数，但现在已经较少使用，本书不再介绍。

2. 字符和字符串

字符或字符串常量是用英文缩略号（形态上很像单引号，一般也就称为单引号）或双引号括起来的单个字符或多个字符，其数值是每个字符对应的 ASCII 码值。例如：'d'（=64H）、'Hello, Assembly !'。在支持汉字的系统中，也可以括起汉字，每个汉字则是两个字节，为汉字机内码或 Unicode。

如果字符串中有单引号本身，可以用双引号；反之亦然，例如：

“Let’s have a try.”

‘Say “Hello”, my baby.’

也可以直接用单引号或者双引号的 ASCII 值（单引号：27H，双引号：22H）。

3. 符号常量

符号常量使用一个符号表达数值。常量若使用有意义的符号名来表示，可以提高程序的可读性，同时更具有通用性；程序中可以多次使用符号常量，但修改时只需改变一处。例如高级语言中就把常用的数值定义为符号常量并保存为常量定义文件，通过包含该文件，程序中就可以直接使用它们。MASM 汇编语言当中也可以如此应用。

MASM 提供的符号定义伪指令有“等价 EQU”和“等号=”。它们用来为常量定义符号名，格式为：

符号名 equ 数值表达式
 符号名 equ <字符串>
 符号名 =数值表达式

等价伪指令 EQU 给符号名定义一个数值或定义成另一个字符串，这个字符串甚至可以是一条处理器指令。例如：

```
NULL equ 0
CR=13
LF=10
CallDOS equ <int 21h>
```

EQU 用于数值等价时不能重复定义符号名，但“=”允许重复赋值。例如：

```
count=100
count=count+64h
```

如果使用“COUNT EQU COUNT+64H”则是错误的。

4. 数值表达式

数值表达式是指用运算符(MASM 统称为操作符 Operator)连接各种常量所构成的算式。汇编程序在汇编过程中计算表达式，最终得到一个确定的数值，所以也属于常量。由于表达式是在程序运行前的汇编阶段计算，所以组成表达式的各部分必须在汇编时就能确定。汇编语言支持多种运算符，但主要应用算术运算符：+（加）、-（减）、*（乘）、/（除）和 MOD（取余数）。当然还可以运用圆括号表达运算的先后顺序。

MOD 进行除法取余数，例如：“10 MOD 4”的结果是“2”。

对于整数数值表达式或地址表达式，参加运算的数值和运算结果必须是整数，除法运算的结果只有商没有余数。地址表达式只能使用加减，常用“地址+常量”或“地址-常量”形式指示地址移动常量表示的若干个存储单元，注意存储单元的单位为字节。

【例 2-1】 数据表达程序。

			;数据段
0000	64 64 64 64 64	const1	db 100,100d,01100100b,64h,'d'
0005	01 7F 80 80 FF FF	const2	db 1,+127,128,-128,255,-1
000B	69 97 20 E0 32 CE	const3	db 105,-105,32,-32,32h,-32h
0011	30 31 32 33 34 35	const4	db '0123456789', 'abcxyz', 'ABCXYZ'
	36 37 38 39 61 62		
	63 78 79 7A 41 42		
	43 58 59 5A		
0027	0D 0A 24	crlf	db 0dh,0ah,'\$'
= 000A		minint	= 10
= 00FF		maxint	equ 0fff
002A	0A 0F FA F5	const5	db minint,minint+5,maxint-5,maxint-minint
002E	10 56 15 EB	const6	db 4*4,34h+34,67h-52h,52h-67h
= int 21h		CallDOS	equ <int 21h>
			;代码段
0017	BA 0011 R	mov dx,offset const4	;从 CONST4 开始显示
001A	B4 09	mov ah,09h	
001C	CD 21	CallDOS	

本例题程序用于说明各种数据的表达形式，使用了定义字节变量 DB 伪指令。左边是列表

文件内容，右边才是源程序本身（编辑源程序文件时，不要把左边列表文件内容录入，下同）。

数据段第一行用不同进制和形式表达了同一个数值：100（=64H），从这一行左边列表文件的5个“64”可以体会到：无论在源程序中如何表达，在计算机内部都是二进制编码。

随后两行给出一些典型数据，用于对比。例如，真值255和-1的机器代码（8位、字节量）都是FFH，128和-128都变换为80H，原因在于它们采用不同的编码，前者是无符号数，后者是补码表达的有符号数。从第3行看出105的补码是69H，-105的补码是97H。（你能看出-32，-32H的补码分别是什么吗？）

第4行定义字符串，对应左边列表文件内容是每个字符的ASCII码值。

随后定义两个数值0DH和0AH，它分别是ASCII表中的回车符和换行符，注意前导零不能省略（否则成为DH和AH，与两个8位寄存器重名），字符“\$”表示了字符串结尾，调用显示功能时需要它。

符号常量MININT数值为10，MAXINT为255，它们只是一个符号，并不占主存空间，应用时可直接将其代表的内容替代。

接着CONST6用表达式定义，但实质还是一个常量；例如，表达式“4*4”计算后为16，对应列表内容是10（表示十六进制10H，即十进制16）。

代码段从CONST4开始显示，遇到字符“\$”结束，所以程序运行后的显示结果是：

```
0123456789abcxyzABCXYZ
```

前一章已经介绍了汇编语言的开发过程，以后各章的程序请读者上机实践，逐渐熟练掌握MASM汇编语言程序的开发方法。汇编过程中，建议生成列表文件；连接生成可执行文件后，要运行文件并查看结果，获得直观的感受。配合查阅列表文件、观察运行结果，本书中的许多解释就更容易理解和掌握了。相信有过高级语言编程经历的读者都有深刻的体会，通过源程序编辑、编译（汇编）、链接，以及可执行文件运行、错误排除和调试等一系列上机实践，很多问题会迎刃而解，自己也常有恍然大悟的感觉，看似艰涩难懂、长篇大论的说明便也一目了然。

2.3 变量应用

程序运行中有很多随之发生变化的结果，需要在可读可写的主存开辟存储空间进行保存，这就是变量（Variable）。变量实质上是主存单元的数据，因而可以改变。变量需要事先定义（Define）才能使用，并具有属性方便应用。

在使用像C、C++和Java等高级语言进行编程时，我们可以使用不同数据类型的变量（例如，字符型、整型或者浮点型），一般不太关心它们在计算机内部的表达和存储。但在机器底层时，我们必须关注数据如何保存，例如有时需要将数据从一种表达形式转换为另一种表达形式。所以，学习汇编语言的变量时请注意本书提供的列表文件，体会数据的存储。

2.3.1 变量定义

变量定义是可以给变量申请固定长度的存储空间，还可以将相应的存储单元初始化。

1. 变量定义伪指令

变量定义伪指令是最常使用的汇编语言说明性语句，它的汇编语言格式为：

变量名 变量定义伪指令 初值表

变量名即汇编语句名字部分，是用户自定义的标识符，表示初值表首个数据的逻辑地址。汇编语言使用这个符号表示地址，故有时被称为符号地址。变量名可以没有，这种情况，汇编程序将直接为初值表分配空间，无符号地址。设置变量名是为了方便存取它指示的存储单元。

初值表是用逗号分隔的参数，由各种形式的常量和特殊的符号“?”、“DUP”组成。其中“?”表示初值不确定，即未赋初值。多个存储单元如果初值相同，可以用复制操作符DUP进行说明。DUP的格式为：

重复次数 dup(重复参数)

变量定义伪指令有DB、DW、DD、DF、DQ和DT（MASM 6.0开始还对应支持BYTE、WORD、DWORD、FWORD、QWORD和TBYTE，两者功能相同），它们根据申请的主存空间单位分类，如表2-5所示。不同的变量类型在表达整数时，只是使用的二进制位数（长度）不同，能够表达的数据范围不同，但有符号数都采用补码编码。

表 2-5 变量定义伪指令

助 记 符	变 量 类 型	变量定义功能
DB	字节（BYTE）	分配一个或多个字节单元；每个数据是字节量，也可以是字符串常量 字节量表示 8 位无符号数或有符号数，字符的 ASCII 码值
DW	字（WORD）	分配一个或多个字单元；每个数据是字量、16 位数据 字量表示 16 位无符号数或有符号数、16 位段选择器、16 位偏移地址
DD	双字（DWORD）	分配一个或多个双字单元；每个数据是双字量、32 位数据 双字量表示 32 位无符号数或有符号数、32 位段基地址、32 位偏移地址
DF	3 个字（FWORD）	分配一个或多个 6 字节单元；6 字节量常表示含 16 位段选择器和 32 位偏移地址 的 48 位指针地址
DQ	4 个字（QWORD）	分配一个或多个 8 字节单元；8 字节量表示 64 位数据
DT	10 个字节（TBYTE）	分配一个或多个 10 字节单元，表示 BCD 码、10 字节数据（浮点处理单元支持）

除了 DB、DW、DD 等定义的简单变量，汇编语言还支持复杂的数据变量，例如结构（Structure）、记录（Record）、联合（Union）等。

2. 字节量数据

用 DB 定义的变量是 8 位、字节量（Byte-sized）数据（对应 C、C++语言的 char 类型）。它不仅可以表示无符号整数 0~255，补码表示的有符号整数-128~+127，一个字符（ASCII 码值），还可以表达压缩 BCD 码 0~99，非压缩 BCD 码 0~9 等。

【例 2-2】 字节变量程序。

			;数据段
= 000A		minint	= 10
0000	00 80 FF 80 00 7F	bvar1	db 0,128,255,-128,0,+127
0006	01 FF 26 DA 38 C8	bvar2	db 1,-1,38,-38,38h,-38h
000C	00	bvar3	db ?
000D	0005 [bvar4	db 5 dup ('\$')
	24		
]		
0012	000A [bvar5	db minint dup(0),minint dup(minint,?)
	00		
]		

```

000A [
0A 00
]
0030 0002 [      db 2 dup(2,3,2 dup(4))
02 03
0002 [
04
]
]

```

在数据表达的例 2-1 中已经应用到不少字节定义变量，通过本例可以再次观察并深入理解。本例的程序重点说明无初值和重复初值的情况。

变量 BVAR3 无初值，表示在主存为该变量保留相应的存储空间。既然有存储空间，就一定有内容，但内容应是任意、不定的；而事实上汇编程序是用 0 填充的（像高级语言的编译程序一样）。

本例的程序通过 DUP 操作符为 BVAR4 定义了 5 个相同的数据，在左侧列表文件中用中括号表示。DUP 操作符可以嵌套，像最后一个无变量名的变量初值依次是：02 03 04 04 02 03 04 04。

3. 字量数据

用 DW 定义的变量是 16 位、字量（Word-sized）数据（对应 C、C++ 语言的 short 类型）。字量数据包含高低两个字节，可以表示更大的数据。8086 逻辑地址的段地址和偏移地址都是 16 位的，可以用 16 位变量保存。

【例 2-3】 字变量程序。

			;数据段	
= 000A		minint	= 10	
0000	0000 8000 FFFF	wvar1	dw 0,32768,65535,-32768,0,+32767	
	8000 0000 7FFF			
000C	0001 FFFF 0026	wvar2	dw 1,-1,38,-38,38h,-38h	
	FFDA 0038 FFC8			
0018	0000	wvar3	dw ?	
001A	2010 1020	wvar4	dw 2010h,1020h	
001E	0005 [dw 5 dup(minint,?)	
	000A 0000			
]			
0032	3139 3832	wvar6	dw 3139h,3832h	
0036	39 31 32 38	bvar6	db 39h,31h,32h,38h	
003A	24		db '\$'	
			;代码段	
0017	BA 0032 R	mov dx,offset wvar6		;从 WVAR6 开始显示
001A	B4 09	mov ah,09h		
001C	CD 21	int 21h		

每个 DW 伪指令定义的变量数据都是 16 位的，所以真值 1 和 -1，用字节量表达是 01H 和 FFH，用字量表达则是 0001H 和 FFFFH。对于有符号数据，最高位仍是符号位，负数真值 -38H，对应补码 FFC8H（=[1]0000H-0038H）。

16 位为两个字节，在以字节为基本存储单元的处理器主存中占用两个连续的存储单元。例如，同样是无初值定义变量，前一个示例的 BVAR3 占 1 字节，本示例的 WVAR3 占两个字节。那么高低两个字节是怎样存放在主存的两个存储单元的呢？是低字节数据存放在低地址存储单元、高字节数据存放在高地址存储单元，还是低字节存放在高地址、高字节存放在低地址？包括 8086 的 80x86 系列处理器采用前者，即所谓的“低对低、高对高”，被称为小端方式（Little Endian）；有些处理器采用后者，被称为大端方式（Big Endian）。列表文件左侧将字变量 WVAR6 两个初值以字量数据显示，按日常规律高位在前、低位在后，分别是 3139 和 3832。而字节变量 BVAR6 仍以字节量形式显示，所以两者存放的内容相同，如图 2-5 所示，也可以从本程序执行后屏幕显示结果（91289128）看出。

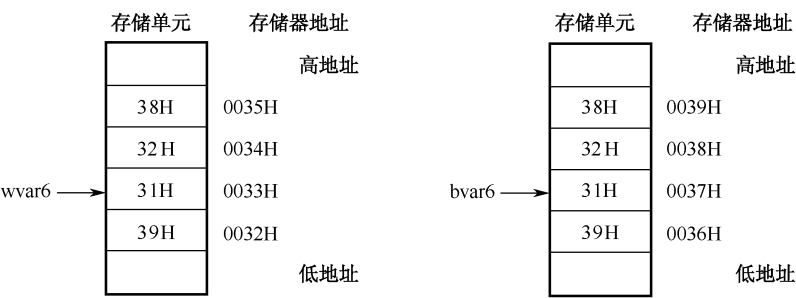


图 2-5 数据的存放顺序

4. 双字量数据

用 DD 定义的变量是 32 位、双字量（Doubleword-Sized）数据（对应 C、C++语言的 long 类型），占用 4 个连续的字节空间，采用小端方式存放。

【例 2-4】 双字变量程序。

```

;数据段
minint = 10
dvar1 dd 0,80000000h,0fffffffh,-80000000h,0,7fffffffh

0000 00000000
      80000000
      FFFFFFFF
      80000000
      00000000
      7FFFFFFF

0018 00000001 dvar2 dd 1,-1,38,-38,38h,-38h
      FFFFFFFF
      00000026
      FFFFFFFDA
      00000038
      FFFFFFFC8

0030 00000000 dvar3 dd ?
0034 00002010 00001020 dd 2010h,1020h
003C 000A [ dvar5 dd minint dup(minint,?)
      0000000A 00000000
      ]

008C 38323139 dvar6 dd 38323139h

```

0090	39 31 32 38	bvar6	db 39h,31h,32h,38h	
0094	24		db '\$'	
			;代码段	
0017	BA 008C R		mov dx,offset dvar6	;从 DVAR6 开始显示
001A	B4 09		mov ah,9	
001C	CD 21		int 21h	

本例题程序定义的数据 DVAR2 似乎与例 2-3 程序中的 WVAR2 一样，但由于采用了双字类型，所以同样的数据却占用 4 字节。术语小端（Little Endian）和大端（Big Endian）来自格利佛游记（Gulliver's Travels）中的小人国故事，小人儿们为吃鸡蛋从小端打开还是从大端打开发起了一场“战争”。专家在制定网络传输协议时借用了这两个词汇，这就是计算机结构中的字节顺序问题，在多字节数据的传输、存储和处理中都存在这样的问题。就像吃鸡蛋无所谓小端还是大端，两种字节顺序形式各有特点，都不比对方更好。只是有些情况更适合小端方式，有些情况采用大端方式更快。例如 Intel 公司采用小端方式，而大多数精简指令集（RISC）计算机则采用大端方式。

8086 处理器采用“低对低、高对高”的小端方式，如果从偏移地址 0090H 开始连续 4 个存储单元的内容依次是 39H、31H、32H、38H，如图 2-6 所示，那么，存储地址 0090H 处的字节量是 39H，0090H 处的字量是 3139H，0090H 处的双字量是 38323139H。理解了这些，看到本示例程序运行后显示“91289128”，就不奇怪了。

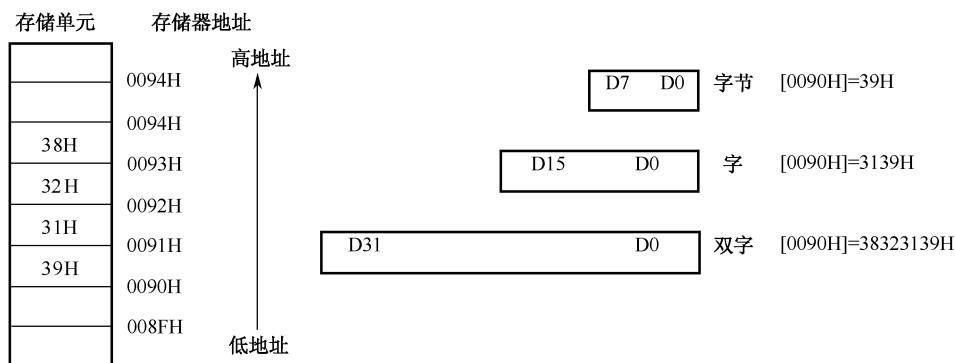


图 2-6 小端存储方式

5. 变量定位

变量定义的存储空间是按照书写的先后顺序一个接一个分配的。而定位伪指令可以控制其存放的偏移地址。

(1) ORG 伪指令

ORG 伪指令将参数表达的偏移地址作为当前偏移地址，格式是：

org 参数

例如，从偏移地址 100H 处安排数据或程序，可以使用语句：

org 100h

(2) ALIGN 伪指令

对于以字节为存储单位的主存储器来说，多字节数据不仅存在按小端或大端方式存放问题，还存在是否对齐地址边界问题。

对 N 个字节的数据 ($N=2, 4, 8, 16, \dots$)，如果起始于能够被 N 整除的存储器地址

位置（也称为模 N 地址）存放，则对齐地址边界。例如，16 位 2 字节数据起始于偶地址（模 2 地址，地址最低 1 位为 0），32 位 4 字节数据起始于模 4 地址（地址最低 2 位为 00）就是对齐地址边界。

难道不允许 N 字节数据起始于非模 N 地址吗？

有很多处理器要求数据存放必须对齐地址边界，否则会发生非法操作。而 8086 处理器比较灵活，允许不对齐边界存放数据。不过，访问未对齐地址边界的数据，处理器需要更多的读写操作，性能不到对齐地址边界的数据访问，尤其需要进行大量频繁的存储器数据操作时。

所以，为了获得更好的性能，常要进行地址边界对齐。ALIGN 伪指令便是用于此目的，其格式如下：

`align n`

其中， N 是对齐的地址边界值，取 2 的乘方（2，4，8，16，...）。另外，EVEN 伪指令实现对齐偶地址，与“ALIGN”语句功能一样。不过 ALIGN 伪指令的 N 值要小于所在段的定位属性值。例如，使用简化段定义格式设置的 16 位的代码段和数据段定位属性是字（Word），其定位属性值是 2（字节）。

【例 2-5】 变量定位程序。

```
                ;数据段
                org 100h
0100 64        bvar1    db 100
                align 2
0102 0064      wvar2    dw 100
                even
0104 00000000   dvar3    dd ?
```

通过列表文件可以看到，汇编程序将 BVAR1 安排在 0100H 相对地址，这是 ORG 伪指令指示的地址，否则作为第一个变量通常起始于 0。

BVAR1 之后接着的存储单元是 0101H，但“ALIGN 2”语句指示对齐模 2 地址（偶地址），所以 WVAR2 被安排在 0102H。

同样，“EVEN”语句指示对齐模 2 地址，所以 DVAR3 占用 0104H~0107H 地址单元。

指令代码也由汇编程序按照语句的书写顺序安排存储空间，定位伪指令也同样可以用于控制其偏移地址。但注意顺序执行的指令之间不能使用对齐伪指令 ALIGN（和 EVEN）。例如，在微型 TINY 存储模式下，.STARTUP 指令实际上就被汇编成为“ORG 100H”语句。这时因为，MS-DOS 要求 COM 格式的可执行程序必须预留前 256（=100H）字节空间用于存放所谓的“程序段前缀 PSP”。

另外，在设置了存储模式后，逻辑段的顺序默认是采用标准 DOS 程序顺序，即地址从低到高依次安排：代码段、数据段和堆栈段。各逻辑段之间也有默认的边界定位规定，一个段不必紧接着另一个段（中间可能有未用空间），逻辑段起始的偏移地址不一定是 0。

2.3.2 变量属性

变量定义除分配存储空间和赋初值外，还可以创建变量名。这个变量名一经定义便具有两类属性：

- （1）地址属性——指首个变量所在存储单元的逻辑地址，含有段基地址和偏移地址。
- （2）类型属性——指变量定义的数据单位，有字节量、字量、双字量、3 字量、4 字量

和 10 字节量，依次用类型名 BYTE、WORD、DWORD、FWORD、QWORD 和 TBYTE 表示。
在汇编语言程序设计中，经常会用到变量名的属性，因此汇编程序提供有关的操作符，以方便获取这些属性值，如表 2-6 所示。

表 2-6 常用的地址和类型操作符

属 性	操 作 符	作 用
地址	[]	将括起的表达式作为存储器地址指针
	\$	返回当前偏移地址
	OFFSET 变量名	返回变量名所在段的偏移地址
	SEG 变量名	返回段地址（高 16 位）
类型	类型名 PTR 变量名	将变量名按照指定的类型使用
	TYPE 变量名	返回一个字量数值，表明变量名的类型
	LENGTHOF 变量名	返回整个变量的数据项数（即元素数）
	SIZEOF 变量名	返回整个变量占用的字节数

1. 地址操作符

地址操作符用于获取变量名的地址属性，主要有 SEG 和 OFFSET，分别取得变量名的段地址和偏移地址两个属性值。中括号和美元符与地址有关，也可以归类为地址操作符。

【例 2-6】 变量地址属性程序。

```
                                ;数据段
0000      12 34                bvar      db 12h,34h
                                org $+10
000C      0001 0002 0003 0004 arraydw 1,2,3,4,5,6,7,8,9,10
                                0005 0006 0007
                                0008 0009 000A
5678 9ABC                                wvar dw 5678h,9abch
0024      = 0018                arr_size=$-array
= 000C                                arr_len  =arr_size/2
0024      DEF0                  dvar dd 9abcdef0h
                                ;代码段
0017      A0 0000 R              mov al,bvar
001A      8A 26 0001 R           mov ah,bvar+1
001E      8B 1E 0022 R           mov bx,wvar[2]
0022      B9 000C                mov cx,arr_len
0025      BA 0025 R              mov dx,$
0028      BE 0020 R              mov si,offset wvar
002B      8B 3C                  mov di,[si]
002D      8B 2E 0020 R           mov bp,wvar
```

列表文件总是将段开始的偏移地址假设为 0（但并不表示主存中其偏移地址一定是 0），然后计算其他数据或代码的相对偏移地址。头一个字节变量 BVAR 有两个数据，占用 0000H 和 0001H 存储单元。

操作符“\$”代表当前偏移地址值，即前一个存储单元分配后当前可以分配的存储单元的偏移地址。语句“ORG \$+10”表示在当前偏移地址（=0002H）基础上加 10，即跳过 10 个字节空间，然后安排变量 ARRAY；所以其偏移地址为 000CH（=2+10）。再分配 12 个字

变量数据后，当前相对偏移地址成为 0024H。所以，符号常量 `ARR_SIZE=0018H (=0024H-000CH)`，即 `ARRAY` 和 `WVAR` 变量所占存储空间的字节数 `18H=24 ([10+2]×2)`。因为每个字变量值占 2 字节，故 `ARR_LEN` 等于它们的数据项数（个数）：`CX=0000000CH=24÷2=12`。

变量名具有逻辑地址。数据段中直接使用变量名就代表它的偏移地址（也可以加个 `OFFSET` 以示明确）。程序代码中，通过引用变量名指向其首个数据，通过变量名加减常量存取以首个数据为基地址的前后数据。`BVAR` 表示它的头一个数据，故 `AL=12H`；`BVAR+1` 表示下一个字节的数据，故 `AH=34H`。变量名实际上就是用地址操作符“`[]`”括起变量名所代表的偏移地址。变量名后用“`+n`”或“`[n]`”作用相同，都表示后移 n 字节存储单元。所以 `WVAR[2]` 指 `WVAR` 两个字节之后的数据：`BX=9ABCH`。

代码段中通过“`$`”获得当前指令“`MOV DX,$`”的偏移地址传送给 `DX`。

语句“`MOV SI,OFFSET WVAR`”通过 `OFFSET` 操作符获得字变量 `WVAR` 的偏移地址，传送给 `SI`。“`[SI]`”则指示该偏移地址的存储单元，从中获取一个字数据正是 `WVAR` 指向的首个变量值；而指令“`MOV BP,WVAR`”也将使 `BP` 等于 `WVAR` 首个变量值。

注意，程序的数据段和代码段开始的实际偏移地址不一定是 0。所以，本例中 `SI` 和 `DX` 并不一定是列表文件的相对偏移地址 20H 和 25H。

操作系统没有提供显示寄存器内容的功能调用，为了观察到程序执行后各个寄存器的结果，可以使用本书配套的输入输出子程序库。读者可以参看例 1-1a 程序（`eg101a.asm`），在本例程序开始加入包含伪指令“`INCLUDE IO.INC`”，在最后调用显示 16 位通用寄存器子程序 `DISPRW` 显示 8 个 16 位通用寄存器内容（参见附录 C），指令是：`CALL DISPRW`。

程序运行结果是（数值采用十六进制表示）：

`AX=3412 BX=9ABC CX=000C DX=0025 SI=0020 DI=5678 BP=5678 SP=0430`

2. 类型操作符

类型操作符使用变量名的类型属性。与大多数程序设计语言一样，在汇编语言中变量也需要先定义，并给定一种类型，每个变量通常表示相应类型的数值。类型转换操作符 `PTR` 用于更改变量名的类型，以满足指令对操作数的类型要求。“类型名”可以是 `BYTE`，`WORD`，`DWORD`，`FWORD`，`QWORD` 和 `TBYTE`（依次表示字节、字、双字、3 字、4 字和 10 字节），还可以是由结构、记录等定义的类型。

`MASM` 中，各种变量类型在 16 位平台用一个字量数值（在 32 位平台则是双字量数值）表达，这就是 `TYPE` 操作符取得的数值。对变量，`TYPE` 返回该类型变量一个数据项所占的字节数，例如对字节、字和双字变量依次返回 1、2 和 4。`TYPE` 后跟常量和寄存器名，则分别返回 0 和该寄存器所能保存数据的字节数。因为常量没有类型，寄存器具有类型（8 位和 16 位寄存器分别是字节和字类型，分别返回 1 和 2）。

对变量，还可以用 `LENGTHOF` 操作符获知某变量名指向多少个数据项，用 `SIZEOF` 操作符获知它共占用多少字节空间，即 `SIZEOF 值=TYPE 值×LENGTHOF 值`。对于字节变量和 `ASCII` 字符串变量，`LENGTHOF` 和 `SIZEOF` 的结果相同。

【例 2-7】 变量类型属性程序。

;代码段

```
0017      A1 0000 R   mov ax,word ptr bvar
001A      BB 0001     mov bx,type bvar
```

001D	B9 0002	mov cx,type wvar
0020	BA 0002	mov dx,type array
0023	BE 000A	mov si,lengthof array
0026	BF 0014	mov di,sizeof array
0029	BD 0018	mov bp,arr_size

本例程序采用与上例程序同样的数据段，这里只列出了代码段。

在指令“MOV AX,WORD PTR BVAR”中，AX 是 16 位寄存器，属于字量类型，变量 BVAR 被定义为字节量，两者类型不同；而 MOV 指令不允许不同类型的数据进行传送；所以利用 PTR 改变 BVAR 的类型，结果是将 BVAR 前两个字节数据按照小端方式组合成字量数据传送给 AX (=3412H)。随后的指令利用类型操作符获取相关数值。

如果包含有本书配套的输入输出子程序库，可以在最后增加一条调用 16 位通用寄存器显示子程序的语句（CALL DISPRW），就可以显示程序运行结果如下：

AX=3412 BX=0001 CX=0002 DX=0002 SI=000A DI=0014 BP=0018 SP=0430

除变量名外，还有段名、子程序名等伪指令的名字以及硬指令的标号，它们也都有地址和类型属性，也都可以使用地址和类型操作符，将在后续章节介绍。

2.4 数据寻址方式

本章前面介绍了汇编语言如何用常量和变量表达数据，现在学习使用处理器指令如何访问这些数据。指令由操作码和操作数两部分组成，本节学习操作数如何表示和访问。虽然有些指令不需要操作数，但大多数指令都有一个或两个操作数。

笼统地说，数据来自主存或外设。但这个数据可能事先已经保存在处理器的寄存器中，也可能与指令操作码一起进入了处理器。主存和外设在汇编语言当中被抽象为存储器地址或 I/O 地址，寄存器通常以名称表达，但在机器代码中同样用地址编码区别寄存器，所以指令的操作数需要通过地址指示。这样，通过地址才能查找到数据本身，这就是数据寻址方式（Data-Addressing Mode）。对处理器的指令系统来说，绝大多数指令采用相同的寻址方式。寻址方式对处理器工作原理和指令功能的理解，以及汇编语言程序设计都至关重要。

汇编语言中，操作码用助记符表示，操作数则由寻址方式体现。8086 处理器只有输入输出指令与外设交换数据，这将在 3.1 节学习。除外设数据外的数据寻址方式有 3 类：

- ⊙ 用常量表达的具体数值（立即数寻址）
- ⊙ 用寄存器名表示的其中内容（寄存器寻址）
- ⊙ 用存储器地址代表保存的数据（存储器寻址）

2.4.1 立即数寻址

立即数寻址（或立即寻址）中，指令需要的操作数紧跟在操作码之后作为指令机器代码的一部分，并随着处理器的取指操作从主存进入指令寄存器。这种操作数用常量形式直接表达，从指令代码中立即得到，被称为立即数（Immediate），如图 2-7 左侧所示。立即数寻址方式只用于指令的源操作数，在传送指令中常用来给寄存器和存储单元赋值。

例如，将数据 0102H 传送到 AX 寄存器的指令，可以书写为：

mov ax,0102h ;指令代码：B8 02 01

这个指令的机器代码（十六进制）是“B8 02 01”，其中头一个字节是操作码 B8H，后面

2 字节就是立即数本身 0102H。8086 处理器规定数据高字节存放于存储器高地址单元、数据低字节存放于低地址单元，如图 2-7 右侧所示。

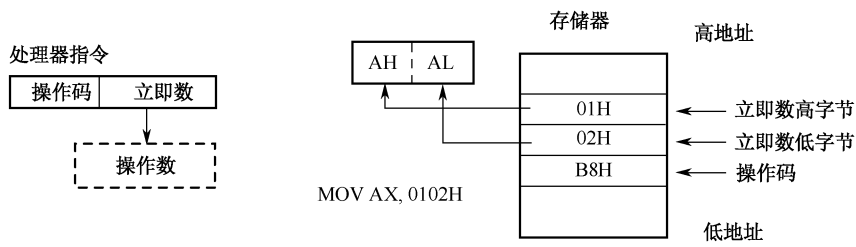


图 2-7 立即数寻址方式

【例 2-8】 立即数寻址程序。

```

                                ;数据段
                                const=64
                                = 0040
0000      87 49                bvar db 87h,49h
0002      1234 04D2            wvar dw 1234h,1234

                                ;代码段
0017      B0 12                mov al,12h
0019      B4 64                mov ah,'d'
001B      BB FFFF            labl: mov bx,-1
001E      B9 0040            mov cx,const
0021      BA 0080            mov dx,const*4/type wvar
0024      BE 0000 R          mov si,offset bvar
0027      BF 001B R          mov di,labl
002A      C6 06 0000 R 4C    mov bvar,01001100b
002F      C7 06 0004 R 0012  mov wvar+2,12h
```

本例程序的代码段所有 MOV 指令的源操作数均采用立即数寻址方式，立即数寻址也只能用于源操作数。它们尽管有多种形式，但汇编后都是一个确定的数值，即立即数。列表文件紧跟着操作码的就是计算机内部对这些立即数的编码。

前 5 条指令使用常量的不同形式表达立即数，依次是十六进制常数、字符（实际就是 ASCII 码值）、十进制负数（内部采用补码）、符号常量（表示其等价的数值）和表达式（其中还使用了类型操作符）。

用地址操作符 OFFSET 获得变量地址，也是立即数寻址方式；标号 LABL 也有地址属性，指令直接使用它就是表示其偏移地址（也可以加上 OFFSET 操作符）。例 2-7 程序中，除第一条指令外，其他指令的源操作数也都是立即数寻址。

立即数也可以传送给变量，后面两条 MOV 指令是示例。

注意，立即数（常量）没有类型，它的类型取决于另一个操作数的类型。所以代码段第一条 MOV 指令的 8 位寄存器 AL，确定“12H”是一个字节量，指令进行 8 位数据传送。最后一条 MOV 指令的变量 WVAR 是字类型，这里的“12H”则是字量（可以看做前导 0 被省略了）。

程序最后可以加入调用子程序 DISPRW 语句，实现 16 位通用寄存器显示，结果如下：

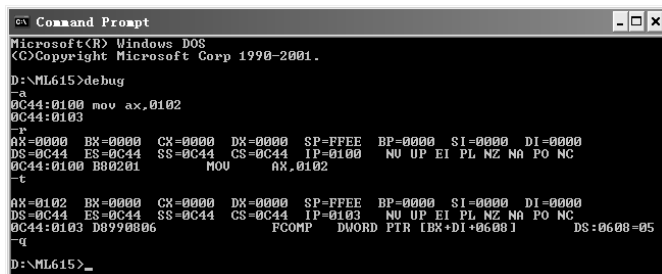
```
AX=6412 BX=FFFF CX=0040 DX=0080 SI=0004 DI=001B BP=091E SP=0410
```

如果还希望更加直观地观察到每条指令的执行，可以利用调试程序（Debugger），它被调试程序提供了一个可以控制和观察的运行环境。在 DOS 平台，可以使用 DEBUG.EXE

调试程序。读者可以通过附录 A 对 DEBUG 调试程序对此有个简单了解，然后结合本章的数据寻址和下一章的常用指令，熟悉汇编、执行程序片段的方法，在第 4 章和第 5 章再掌握可执行程序的调试过程。

如果感觉难以掌握调试程序，可以跳过，不会影响继续学习。虽然暂时不上机亲身体会调试程序，但还是建议能够阅读一下这些内容。这将有助于读者加深对指令功能和程序执行的感性认识和深入理解。

图 2-8 演示立即数寻址示例指令“MOV AX,0102H”的调试过程，操作步骤详述如下：



```

C:\ Command Prompt
Microsoft(R) Windows DOS
(C) Copyright Microsoft Corp 1990-2001.

D:\ML615>debug
-
0C44:0100 mov ax,0102
-
0C44:0103
-
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0C44 ES=0C44 SS=0C44 CS=0C44 IP=0100  NO UP EI PL NZ NA PO NC
0C44:0100 B00201      MOV     AX,0102
-
AX=0102 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0C44 ES=0C44 SS=0C44 CS=0C44 IP=0103  NO UP EI PL NZ NA PO NC
0C44:0103 D8790806      FCOMP  DWORD PTR [BX+DI*0608]  DS:0608=05
-
D:\ML615>_

```

图 2-8 立即数寻址的调试截图

(1) 首先进入 DOS 平台（或者 Windows 控制台），然后在 DOS 命令行提示符下输入：DEBUG，然后回车，进入调试程序。DEBUG 调试程序也是命令行操作方式，默认命令提示符是一个短线“-”。

(2) 在 DEBUG 调试程序的提示符下，输入汇编命令 A（Assemble），即输入 A 并回车，调试程序进入汇编状态。屏幕显示当前可用存储器区域的逻辑地址（十六进制），接着输入的指令代码将保存于此。其中偏移地址不是 0，而是默认为 0100H，因为之前的存储区域是 DOS 为应用程序创建的程序段前缀 PSP。段地址在本截图中显示为 0C44H，这个地址是会随着可用的存储区域变化的。

(3) 在显示的逻辑地址之后，输入指令“MOV AX,0102”，然后回车。注意，DEBUG 调试程序中的数值均默认采用十六进制，不能也不需要加上后缀字母“H”。屏幕提示下一条指令将保存的地址，从这个例子的偏移地址 0103H 可以看出，上一条指令占用了 3 字节。本例中不再输入指令，可以再次回车，便退出汇编状态，显示 DEBUG 调试程序的命令提示符“-”。

(4) 为了便于对比指令执行前后的情况，输入寄存器命令 R（Register），即键入 R 并回车，调试程序显示当前寄存器内容。其中 SP=FFEEH，指向逻辑段高端，其余 7 个通用寄存器都是 0。而 4 个段寄存器相同，4 个段都起始于当前可用存储地址。指令指针 IP 等于 0100H，即默认的将要执行的指令偏移地址，而在下一行也提示该指令，即刚才输入的指令“MOV AX,0102H”。后面 NV、UP 等符号表示标志寄存器中各个标志的状态（参见附录 A）。

(5) 细致观察每个指令（或语句）的执行情况，常用的方法是进行单步操作（单步调试），也就是每执行一条指令（或语句）即暂停、显示它的执行结果或状态。输入跟踪（也常称为单步）命令 T（Trace），即输入 T 并回车。调试程序执行当前 CS: IP 指定存储器地址的指令，然后暂停并像寄存器命令一样显示执行“MOV AX,0102H”指令后当前寄存器中的内容。对比前一个寄存器命令的显示内容，发现 AX=0102 正是该指令实现的功能；同时 IP=0103 增加了该指令所占的 3 个字节数，指向下条指令。本例中仅输入了一条指令，所显示的下条指令是调试程序将主存原存储内容作为一条指令代码解释的现象，不必理会。

(6) 完成调试后，应使用退出命令 Q (Quit) 将调试程序关闭，返回 DOS 平台。

读者可以参考附录 A 尝试其他命令，如反汇编命令 U (Unassemble) 查看输入指令的机器代码。当然，最初一定会看到许多感到困惑的现象，不必迷茫、不必灰心，随着学习的逐渐深入，慢慢就会有所认识。这些知识也将在后续章节中进行说明。

另外，读者可能会尝试汇编其他指令，如例 2-8 代码段的指令。不过，请读者理解 DEBUG 调试程序的局限性。例如，其提供的汇编命令 A 很简单，只能支持逐条指令的翻译，只能使用十六进制数，不能输入加引号的字符（可以使用其 ASCII 值），不能输入标识符（可以使用其所在的偏移地址）。所以，读者要意识到，本书的源程序语句采用的是 MASM 汇编程序支持的语法规则，在调试程序中应遵循调试程序规定的语法，好在它们基本相同。

2.4.2 寄存器寻址

寄存器寻址方式的操作数存放在处理器的寄存器中。通常直接使用寄存器名表示它所保存的数据，即寄存器操作数，如图 2-9 所示。绝大多数指令采用通用寄存器寻址（8086 处理器是 16 位的 AX、BX、CX、DX、SI、DI、BP 和 SP，以及 8 位的 AL、AH、BL、BH、CL、CH、DL 和 DH），部分指令支持专用寄存器，例如段寄存器、标志寄存器等。寄存器寻址方式简单快捷，是最常使用的寻址方式。

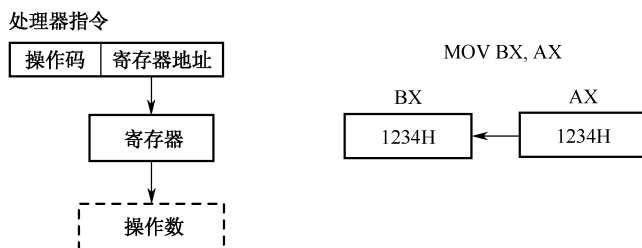


图 2-9 寄存器寻址

在前面示例程序的许多指令中，凡是只使用寄存器名（无其他符号，例如中括号、变量名等）的操作数都为寄存器寻址。

【例 2-9】 寄存器寻址程序。

```
                ;代码段
0017          8A C4      mov al,ah
0019          8B D8      mov bx,ax
001B          8A C8      mov cl,al
001D          8C DA      mov dx,ds
001F          8E C2      mov es,dx
                   mov di,dh ;error!
```

eg209.asm(13) : error A2070: invalid instruction operands

本例程序的指令操作数都是寄存器寻址。寄存器既可以作源操作数，也可以作目的操作数；另一个操作数可以是变量，也可以是常量（源操作数）。但需要注意的是，指令通常要求操作数类型一致，所以最后一条指令“MOV DI,DH”出现错误时，列表文件会将其标明，记录下语句行号（本例是 13 行）、错误编号（本例是 A2070）和错误原因（该指令错误信息的含义是：无效的指令操作数）。汇编程序 MASM 提示的错误信息保存在 ML.ERR 文件中，常见的错误信息见附录 E。

汇编语言程序会因为符号拼写错误、多余的空格、遗忘后缀字母或前导 0、错误的标点、太过复杂的常量或表达式等出现一般的语法错误，也常因为未能熟练掌握指令功能导致操作数类型不匹配、错用寄存器等情况，出现指令语法错误，当然也会因为算法流程、非法地址等出现逻辑错误或者运行错误。注意，汇编程序只能发现语法错误，而且提示的错误信息有时不甚准确，尤其当多种错误同时出现时。应特别留心第一个引起错误的指令，因为后续错误可能因其产生，修改了这个错误也就可能纠正了后续错误。

例如，本例错误的原因更准确地说是两个操作数的类型不匹配，因为 DI 是 16 位寄存器而 DH 是 8 位寄存器，MOV 指令不允许把 8 位寄存器的数据传送到 16 位寄存器中。虽然你会觉得 16 位寄存器可以放下 8 位数据，占用其低 8 位即可。但事实上，8086 处理器的设计师基于简化硬件电路的考虑，没有按照这个思路去设计。所以，这是一条不存在的指令，即非法指令。同时，编写汇编程序的系统程序员也没有人性化地按照这个思路去处理这个问题。对于应用程序员来说，只能遵循这个规则。虽然这个现象让我们很困惑，但是，如果透过现象看本质，我们是否也可略微体会到处理器工作原理和设计思路呢？

2.4.3 存储器寻址

数据很多时候都保存在主存储器中。尽管可以事先将它们取到寄存器中再进行处理，但指令也需要能够直接寻址存储单元进行数据处理。如何寻址主存中存储的操作数就称为存储器寻址方式，也称为主存寻址方式。编程时，存储器地址使用包含段选择器和偏移地址的逻辑地址。

1. 段寄存器的默认和超越

段寄存器（段选择器）有默认的使用规则，如表 2-7 所示。寻址存储器操作数时，段寄存器不用显式说明，即数据就在默认的段中，一般是 DS 段寄存器指向的数据段；如果采用 BP 或 SP 作为基地址指针，则默认使用 SS 段寄存器指向堆栈段。

表 2-7 段选择器的使用规则

访问存储器方式	默认的段寄存器	可超越的段寄存器	偏移地址
读取指令	CS	无	IP
堆栈操作	SS	无	SP
一般的数据访问（下列除外）	DS	CS、ES、SS	有效地址 EA
BP 为基地址的数据访问	SS	CS、ES、DS	有效地址 EA
串指令的源操作数	DS	CS、ES、SS	SI
串指令的目的操作数	ES	无	DI

如果不使用默认的段选择器，需要书写段超越指令前缀显式说明。段超越指令前缀，是一种只能跟随具有存储器操作数的指令之前的指令，其助记符是段寄存器名后跟英文冒号，即 CS:、SS:、DS: 或 ES:。

2. 偏移地址的组成

因为段基地址由默认的或指定的段寄存器指明，所以指令中只要有偏移地址即可。存储器操作数寻址使用的偏移地址常被称为有效地址 EA（Effective Address）。

为了方便各种数据结构的存取，8086 处理器设计了多种主存寻址方式，但可以统一表

达如下（参见图 2-10 所示）：

16 位有效地址=基址寄存器+变址寄存器+位移量

其中基址寄存器是 BX 或 BP，变址寄存器是 SI 或 DI，位移量是 8 或 16 位有符号值。

MASM 表达存储器寻址时，要使用中括号（只有变量名的情况除外）。

基址寄存器

变址寄存器

位移量

$$16 \text{ 位有效地址} = \left\{ \begin{array}{c} \text{BX} \\ \text{BP} \end{array} \right\} + \left\{ \begin{array}{c} \text{SI} \\ \text{DI} \end{array} \right\} + \left\{ \begin{array}{c} 8 \text{ 位位移量} \\ 16 \text{ 位位移量} \end{array} \right\}$$

图 2-10 8086 的存储器寻址

3. 直接寻址

存储器的直接寻址方式的有效地址只有位移量部分，直接包含在指令代码中，如图 2-11（a）所示。直接寻址常用于存取变量。

例如，将变量 COUNT 内容传送给 AX 的指令：

mov ax,count

;也可以表达为：mov ax,[count]

汇编语言的指令代码中直接书写变量名就是在其偏移地址（有效地址）的存储单元读写操作数。假设操作系统为变量 COUNT 分配的有效地址是 2000H，则该指令的机器代码是：A1 00 20，反汇编的指令形式为：MOV AX, DS:[2000H]，其源操作数采用直接寻址。

MASM 汇编程序使用中括号表示偏移地址，变量 COUNT 也可以采用加有中括号 [COUNT] 的形式，体现其访问存储单元的特性。

图 2-11（b）演示了该指令的执行过程（假设 DS 保存段地址 1492H）：该指令代码中数据的有效地址（图中第①步），与数据段寄存器 DS 给定的段地址左移 4 位（图中第②步）后相加，构成操作数所在存储单元的物理地址（图中第③步）。该指令的执行结果是将物理地址 16920H 单元的内容传送至 AX 寄存器（图中第④步）。其中高字节内容送 AH 寄存器，低字节内容送 AL 寄存器。

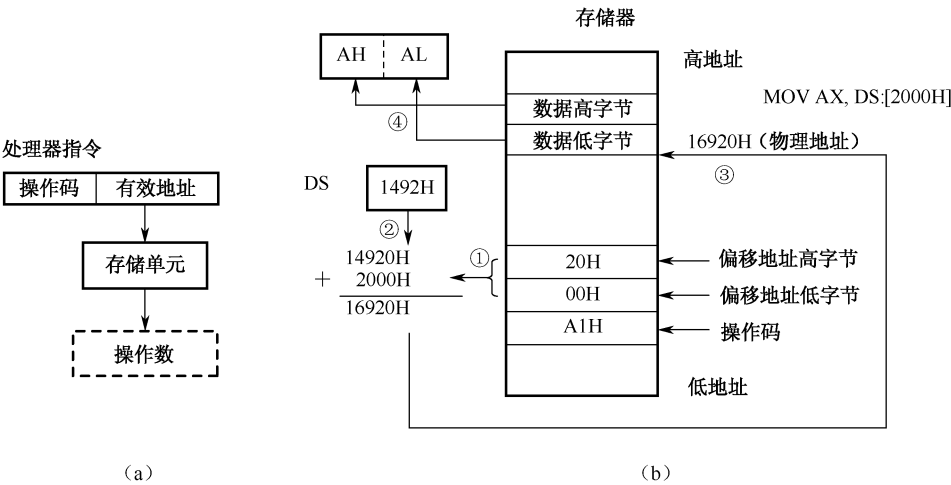


图 2-11 存储器直接寻址

【例 2-10】 存储器直接寻址程序。

0000 87 49

;数据段

bvar db 87h,49h

```

0002      1234 04D2      wvar dw 1234h,1234
                                ;代码段
0017      8A 0E 0000 R   mov cl,bvar
001B      8B 16 0002 R   mov dx,wvar
001F      88 36 0001 R   mov bvar+1,dh
0023      89 16 0004 R   mov byte ptr wvar+2,dl
0027      C7 06 0002 R 4321 mov wvar,4321h
                                mov wvar+2,wvar

```

eg210.asm(14) : error A2070: invalid instruction operands

本例程序的每条指令都采用了直接寻址，或为源操作数或为目的操作数，从列表文件可以看到指令代码中包含其相对地址（后缀 R 是列表文件的一个标志，表示这是一个相对地址，还需要连接时形成真正的偏移地址）。大多数指令不支持两个操作数都是存储器操作数的情况，所以最后一条指令提示有错误，其含义是：无效的指令操作数。

图 2-12 演示直接寻址示例指令“MOV AX,COUNT”的调试过程，操作步骤详述如下：

```

C:\ Command Prompt
Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-2001.

D:\ML615>debug
-a
0C44:0100 mov ax,count
                                ^ Error
0C44:0100 mov ax,[2000]
0C44:0103
-e 2000
0C44:2000 20 12 70 34
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0C44 ES=0C44 SS=0C44 CS=0C44 IP=0100 NU UP EI PL NZ NA PO NC
0C44:0100 A10020      mov     ax,[2000]      DS:2000-3412
-t
AX=3412 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0C44 ES=0C44 SS=0C44 CS=0C44 IP=0103 NU UP EI PL NZ NA PO NC
0C44:0103 D8790806      fcomp  dword ptr [bx+di+0608]  DS:0608-05
-q
D:\ML615>

```

图 2-12 直接寻址的调试截图

(1) 在 DOS 命令行提示符输入 DEBUG，进入调试程序。

(2) 在调试程序提示符下输入汇编命令 A，进入汇编状态。

(3) 在汇编状态下，输入“MOV AX,COUNT”并回车。由于 DEBUG 的汇编命令不支持标识符，所以在下一行用“Error”提示出现错误，之前的符号“^”表示出错的位置。变量的实质是主存单元，可以直接用偏移地址表示，假设该变量保存于数据段 2000H 地址，那么可以输入“MOV AX,2000”指令。本例只输入一条指令，所以按回车便可退出汇编状态。

(4) 在数据段存放变量值，可以使用修改命令 E (Enter)：即输入“E 2000”并回车，其中 2000 表示变量所在地址。调试程序显示指定存储器地址的原保存数据（本例为 20），修改为该变量表示的值：本例输入 12。一个存储单元保存一个字节数据，而本例指令传输一个字即 2 字节数据。所以，按空格键继续输入下一字节内容。同样，调试程序先给出原有数据，用户输入新值。最后回车，退出修改主存内容状态。

(5) 使用寄存器命令 R 观察当前寄存器内容，以及将要执行的指令。注意到该指令所在行最后，调试程序还“友好地”提示了将要访问的主存单元（本例是 DS:2000，表示数据段偏移地址 2000H 位置）的内容，即刚才使用 E 命令输入的数据 3412H。不用感到奇怪，先输入的“12”是低地址字节内容，后输入的“34”是高地址字节内容，按照 8086 的“高对高、低对低”的小端存储方式这个 2 字节的数据确实是 3412H。

(6) 使用跟踪命令 T 单步执行该指令，显示 AX=3412 是该指令执行后的结果。

参阅附录 A，读者还可以使用显示命令 D (Dump) 查看指定主存单元的内容，最后记

得使用退出命令 Q 返回到 DOS。

使用 DEBUG 调试程序时，可能常会遇到莫名其妙的现象或错误，不必担心也不必惊慌，请再次阅读附录 A 中有关指令的说明，特别要留心其中的注意事项，多次尝试就会有收获。

4. 寄存器间接寻址

有效地址存放在寄存器中，就是采用寄存器间接寻址存储器操作数，如图 2-13 (a) 所示。MASM 汇编程序使用英文中括号括起寄存器表示寄存器间接寻址。8086 处理器只有基址寄存器 BX 和两个变址寄存器 SI、DI 可以作为寄存器间接寻址的寄存器。

例如，下面的前两条指令的源操作数、后两条指令的目的操作数都是寄存器间接寻址方式：

```
mov al,[bx]           ;字节量传送, BX 间接寻址
mov cx,[si]           ;字量传送, SI 间接寻址
mov [di],dl           ;字节量传送, DI 间接寻址
mov word ptr [di],1394h ;字量传送, DI 间接寻址
```

寄存器间接寻址中寄存器的内容是偏移地址，相当于一个地址指针。指令“MOV AL, [BX]”执行时，如果 BX=2000H，则该指令功能等同于“MOV AL, DS:[2000H]”，如图 2-13 (b) 所示。

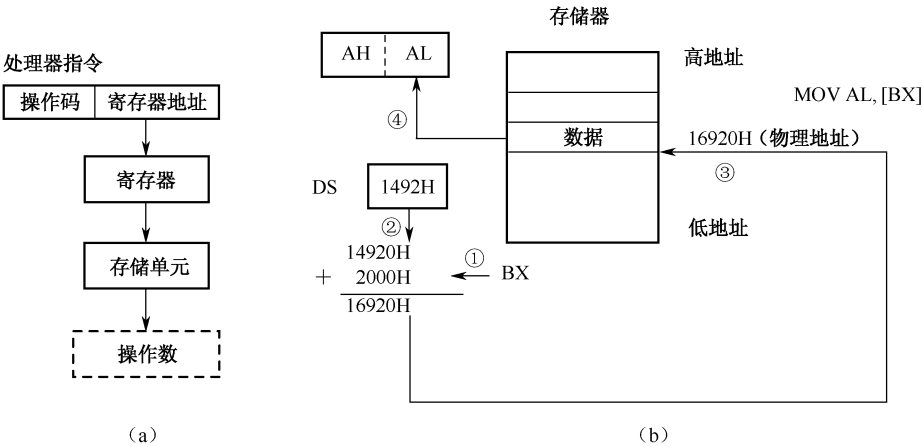


图 2-13 寄存器间接寻址

寄存器间接寻址并没有说明指向的存储单元是何种类型，如果另一个操作数也没有类型，例如立即数，则必须使用类型操作符 PTR 显式说明，正像指令“MOV WORD PTR [DI],1394H”那样。

设计寄存器间接寻址的主要目的之一是可以方便地对数组的元素或字符串的字符进行操作。这只要将数组或字符串首地址（或末地址）赋值给通用寄存器，利用寄存器间接寻址就可以访问到数组或字符串头一个（或最后一个）元素或字符，加减数组元素所占的字节数（对 ASCII 码字符串来说，每个字符占一字节）就可以访问到其他元素或字符。

【例 2-11】 寄存器间接寻址程序。

```
                ;数据段
srcmsg          db 'Try your best, why not.$'
dstmsg          db sizeof srcmsg dup(?)
                ;代码段
```

```

mov cx,lengthof srcmsg      ;CX=字符串字符个数
mov si,offset srcmsg        ;SI=源字符串首地址
mov di,offset dstmsg        ;DI=目的字符串首地址
again: mov al,[si]          ;取源串一个字符送 AL
      mov [di],al          ;将 AL 传送给目的串
      add si,1              ;源串指针加 1, 指向下一个字符
      add di,1              ;目的串指针加 1, 指向下一个字符
      loop again           ;字符个数 CX 减 1, 不为 0, 则转到 AGAIN 标号处执行
      mov dx,offset dstmsg  ;显示目的字符串内容
      mov ah,9
      int 21h

```

本例程序实现将源字符串 SRCMSG 传送给目的字符串 DSTMSG（没有给出列表文件内容，请重点关注程序）。利用 OFFSET 获得源字符串 SRCMSG 首地址传送给 SI，用 SI 寄存器间接寻址访问源字符串的每个字符。同样，目的字符串 DSTMSG 采用 DI 指向，DI 寄存器间接寻址访问每个字符。从源串取一个字符，通过 AL 传送给目的串（MOV 指令不支持两个存储单元直接传送）。接着 SI 和 DI 都加 1（ADD 是加法指令）指向下一个字符，重复传送。循环指令 LOOP（详见 4.3 节）利用 CX 控制计数：首先将 CX 减 1，然后判断 CX 是否为 0，不为 0 则继续循环执行，为 0 则结束。所以程序开始设置 CX 等于字符串的长度（字符个数）。

程序最后显示目标字符串，用于判断是否传送正确（字符串最后的“\$”是结尾字符，不会显示出来）。

5. 寄存器相对寻址

寄存器相对寻址的有效地址是寄存器内容与位移量之和，如图 2-14 所示。8086 只有 BX 和 BP、SI 和 DI 可作为寄存器相对寻址的寄存器。使用 BX、SI 和 DI 寄存器时与前述的直接寻址、间接寻址一样默认访问数据段的数据，而使用 BP 相对寻址默认访问堆栈段的数据；不过它们都可以使用段超越前缀指令改变访问的逻辑段。在 MASM 汇编程序中，只要按照规范书写语句，绝大多数情况都不需要程序员自行书写段超越前缀指令。

例如：

```
mov si,[bx+4]      ;源操作数采用寄存器相对寻址，也可表达为：[4][bx]或 4[bx]
```

这条指令中，源操作数的有效地址由 BX 寄存器内容加位移量 4 获得，默认与 BX 寄存器配合的是 DS 指向的数据段。再如：

```
mov di,[bp-08h]    ;源操作数也可表达为：[-08h][bp]，但不能是：-08h[bp]
```

该指令源操作数的有效地址等于 BP-8，与之配合的默认段寄存器为 SS。

偏移量还可以采用其他常量形式，也可以使用变量所在的地址作为偏移量。例如：

```
mov ax,count[si]   ;源操作数也可表达为：[si+count]，或[count][si]
```

这里用变量名 COUNT 表示其偏移地址用做相对寻址的偏移量，由其地址与寄存器 SI 相加的数据作为有效地址访问存储单元。

像寄存器间接寻址一样，寄存器相对寻址也可以方便地对数组的元素或字符串的字符进行操作。这只要用数组或字符串首地址为位移量，赋值寄存器等于数组元素或字符的所在的

处理器指令

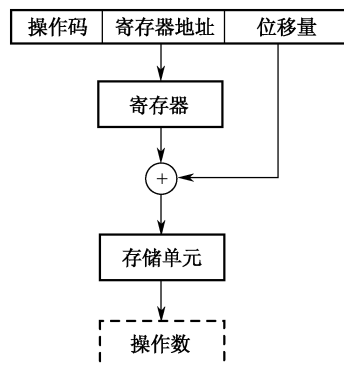


图 2-14 寄存器相对寻址

位置量。

下面例题程序采用寄存器相对寻址完成例 2-11 同样的字符串复制功能。

【例 2-12】 寄存器相对寻址程序。

```
                ;数据段
srcmsg         db 'Try your best, why not.$'
dstmsg         db sizeof srcmsg dup(?)
                ;代码段
                mov cx,lengthof srcmsg    ;CX=字符串字符个数
                mov bx,0                  ;BX 指向首个字符
again:          mov al,srcmsg[bx]         ;取源串一个字符送 AL
                mov dstmsg[bx],al        ;将 AL 传送给目的串
                add bx,1                  ;加 1, 指向下一个字符
                loop again                ;字符个数 CX 减 1, 不为 0, 则转到 AGAIN 标号处执行
                mov dx,offset dstmsg      ;显示目的字符串内容
                mov ah,9
                int 21h
```

本例程序采用寄存器相对寻址，变量所在的偏移地址与寄存器 BX 相加得到有效地址，而访问字符串。对比寄存器间接寻址，采用寄存器相对寻址的程序更简洁一些，但每个相对地址都需要指令进行加法运算（增加了指令的复杂性）。

6. 基址变址寻址

使用基址寄存器内容加上变址寄存器内容形成有效地址寻址存储器操作数被称为基址变址寻址。8086 的基址寄存器是 BX 和 BP，变址寄存器是 SI 和 DI，使用 BP 默认访问堆栈段，其他默认访问数据段，可以使用段超越。

例如：

```
mov al,[bx+si]    ;源操作数在数据段，也可以表达为：[bp][di]
mov ax,[bp+di]    ;源操作数在堆栈段，也可以表达为：[bp][di]
mov ax,ds:[bp+si] ;源操作数在数据段，也可以表达为：ds:[bp][si]
```

7. 相对基址变址寻址

采用相对基址变址寻址，存储器操作数的有效地址由基址寄存器内容、变址寄存器内容及位移量相加获得。同样，8086 的基址寄存器是 BX 和 BP，变址寄存器是 SI 和 DI，使用 BP 默认访问堆栈段，其他默认访问数据段，可以使用段超越。

例如：

```
mov cx,[bx+si+4]    ;源操作数也可以表达为：4[bx+si]
mov ax,80h[bx+di]    ;源操作数也可以表达为：80h[bx][di]
mov dx,count[bp][di] ;源操作数也可以表达为：[bp+di+count]
```

当存储器操作数由 2 或 3 个部分组成时，MASM 允许写入一个中括号内并用加号连接 2 个或 3 个部分，也允许各个部分都使用中括号，但注意基址寄存器要写在变址寄存器之前。位移量可以使用常量或者变量名（使用其偏移地址），也允许不写入中括号内，但这种情况下只能在中括号前、数字开头不能使用正号（+）或负号（-）。

设计基址变址寻址和相对基址变址寻址的主要目的是便于支持二维数组等更复杂的数据结构。

2.4.4 数据寻址的组合

至此，已经学习了绝大多数指令采用的数据寻址方式，下面做一个简单总结，便于在以后的编程实践中掌握它们的具体应用。

(1) 立即数寻址——只能用于源操作数，其类型由另一个操作数的类型或指令决定。本书统一使用 `imm` 符号表示立即数，而 8086 处理器支持 16 位立即数（使用符号 `i16` 表示）和 8 位立即数（使用符号 `i8` 表示）。

(2) 寄存器寻址——主要是指通用寄存器寻址，最常使用、可以单独或同时用于源操作数和目的操作数，寄存器本身包含有类型。本书统一使用符号 `reg` 表示通用寄存器，对 8086 处理器来说有 8 个 16 位通用寄存器 `r16`（`AX`、`BX`、`CX`、`DX`、`SI`、`DI`、`BP` 和 `SP`）和 8 个 8 位通用寄存器 `r8`（`AH`、`AL`、`BH`、`BL`、`CH`、`CL`、`DH` 和 `DL`）。部分指令可以使用专用寄存器，例如段寄存器 `seg`（`CS`、`DS`、`SS`、`ES`）。

(3) 存储器寻址——访问的数据存放在主存，利用逻辑地址指示。段基址由默认或指定的段寄存器指出，指令代码只表达偏移地址，称为有效地址，有多种形式，对应多种存储器寻址方式。本书统一用 `mem` 表示存储器操作数，可以是 16 位或 8 位数据，分别用符号 `m16` 和 `m8` 表示。

典型的指令操作数有两个，一个书写在左边，被称为目的操作数 `DEST`，另一个用逗号分隔书写在右边，被称为源操作数 `SRC`。数据寻址方式在指令中并不可以任意组合的，要遵循规律并符合逻辑。例如，绝大多数指令（数据传送、加减运算、逻辑运算等常用指令）都支持如下组合，如图 2-15 所示：

处理器指令助记符 `reg,imm/reg/mem`
处理器指令助记符 `mem,imm/reg`

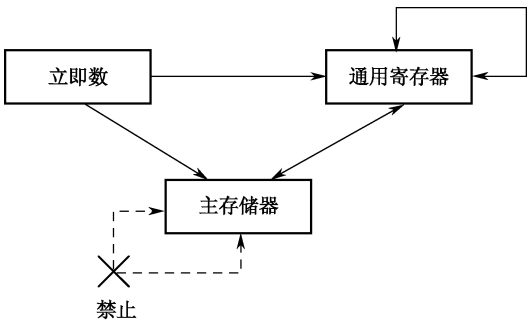


图 2-15 数据寻址的组合

在这两个操作数中，源操作数可以由立即数、寄存器或存储器寻址，而目的操作数只能是寄存器或存储器寻址，并且两个操作数不能同时为存储器寻址方式。

第 3 章开始将陆续介绍 8086 处理器的常用指令，并使用约定符号，参见表 2-8。除特别说明的新符号外，凡不符合指定格式的指令都是不存在的非法指令。附录 F 罗列了全部指令。

高级语言虽然不讨论数据寻址，但实际上其复杂数据类型和构造的数据结构都需要处理器数据寻址的支持，这也是处理器设计多种灵活的访问数据方式的重要原因。

表 2-8 寻址方式及其符号

寻址方式	符号及说明
立即数寻址	imm (包括 8 位立即数 i8, 16 位立即数 i16)
寄存器寻址	通用寄存器 reg (包括 8 位通用寄存器 r8, 16 位通用寄存器 r16), 段寄存器 seg
存储器寻址	mem (包括 8 位存储器操作数 m8, 16 位存储器操作数 m16)

习 题 2

2.1 简答题

- (1) 使用二进制 8 位表达无符号整数, 257 有对应的编码吗?
- (2) 字符 “F” 和数值 46H 作为 MOV 指令的源操作数有区别吗?
- (3) 为什么可以把指令 “MOV AX, (34+67H)*3” 中的数值表达式看成是常量?
- (4) 汇编语言为什么规定十六进制数若以 A~F 开头, 需要在前面加个 0?
- (5) 数值 500, 能够作为字节变量的初值吗?
- (6) 多字节数据对齐地址边界有什么作用?
- (7) 为什么将查找操作数的方法称为数据寻 “址” 方式?
- (8) 为什么变量 VAR 在指令 “MOV AX, VAR” 中表达直接寻址?
- (9) 指令 “MOV AX, [SI]” 从哪个段获得存储器操作数?
- (10) “MOV BX, CL” 是一条正确的指令吗?

2.2 判断题

- (1) 对一个正整数, 它的补码与无符号数的编码一样。
- (2) 常用的 BCD 码为 8421 BCD 码, 其中的 8 表示 D3 位的权重。
- (3) 排序一般按照 ASCII 码值大小, 从小到大升序排列时, 小写字母排在大写字母之前。
- (4) 用 “DB” 和 “DW” 定义变量, 如果初值相同, 则占用的存储空间也一样多。
- (5) “TYPE DX” 的结果是一个常量, 等于 2。
- (6) 8086 处理器采用小端方式存储多字节数据。
- (7) 某个字变量存放于存储器地址 0403H~0404H 中, 对齐了地址边界。
- (8) 立即数寻址只会出现在源操作数中。
- (9) 存储器寻址方式的操作数当然在主存了。
- (10) 指令 “MOV AX, VAR+2” 与 “MOV AX, VAR[2]” 功能相同。

2.3 填空题

(1) 计算机中有一个 “01100001” 编码。如果把它认为是无符号数, 它是十进制数 _____; 如果认为它是 BCD 码, 则表示真值 _____; 如果它是某个 ASCII 码, 则代表字符 _____。

(2) C 语言用 “\n” 表示让光标回到下一行首位, 在汇编语言中需要输出两个控制字符: 一个是回车, 其 ASCII 码是 _____, 它将光标移动到当前所在行的首位; 另一个是换行, 其 ASCII 码是 _____, 它将光标移到下一行。

(3) 定义字节变量的伪指令助记符是 _____, 获取变量名所具有的偏移地址的操作符是 _____。

(4) 数据段有语句 “H8843 DD 99008843H”, 代码段指令 “MOV CX, WORD PTR H8843”

执行后, CX=_____。

(5) 用 DD 定义的一个变量 XYZ, 它的类型是_____, 用 “TYPE XYZ” 会得到数值为_____。如果将其以字量使用, 应该用_____说明。

(6) 数据段有语句 “ABC DB 1,2,3”, 代码段指令 “MOV CL, ABC+2” 执行后, CL=_____。

(7) 除外设数据外的数据寻址方式有 3 类, 分别被称为_____, _____和_____。

(8) 指令 “MOV AX, OFFSET MSG” 的目的操作数和源操作数分别采用_____和_____寻址方式。

(9) 已知 SI=04000H, BX=20H, 指令 “MOV AX, [BX+SI+8]” 中访问的有效地址是_____。

(10) 用 BX 做基地址的指令, 默认采用_____段寄存器指向的数据段; 如果采用 BP 作为基地址指针, 默认使用_____段寄存器指向堆栈段。

2.4 下列十六进制数表示无符号整数, 请转换为十进制形式的真值:

(1) FFH (2) 0H (3) 5EH (4) EFH

2.5 将下列十进制数真值转换为压缩 BCD 码:

(1) 12 (2) 24 (3) 68 (4) 99

2.6 将下列压缩 BCD 码转换为十进制数:

(1) 10010001 (2) 10001001 (3) 00110110 (4) 10010000

2.7 将下列十进制数用 8 位二进制补码表示:

(1) 0 (2) 127 (3) -127 (4) -57

2.8 进行十六进制数的加减运算, 并说明是否有进位或借位:

(1) 1234H+7802H

(2) F034H+5AB0H

(3) C051H-1234H

(4) 9876H-ABCDH

2.9 数码 0~9、大写字母 A~Z、小写字母 a~z 对应的 ASCII 码分别是多少? ASCII 码 0DH 和 0AH 分别对应什么字符?

2.10 设置一个数据段, 按照如下要求定义变量或符号常量:

(1) my1b 为字符串变量: Personal Computer

(2) my2b 为用十进制数表示的字节变量: 20

(3) my3b 为用十六进制数表示的字节变量: 20

(4) my4b 为用二进制数表示的字节变量: 20

(5) my5w 为 20 个未赋值的字变量

(6) my6c 为 100 的常量

(7) my7c 表示字符串: Personal Computer

2.11 定义常量 NUM, 其值为 5; 数据段中定义数组变量 DATALIST, 它的头 5 个字节单元中依次存放 -10, 2, 5 和 4, 最后 1 个单元初值不定。

2.12 从低地址开始以字节为单位, 用十六进制形式给出下列语句依次分配的数值:

db 'ABC', 10, 10h, 'EF', 3 dup(-1, ?, 3 dup(4))

dw 10h,-5,3 dup(?)

2.13 设在某个程序中有如下片段，请写出每条传送指令执行后寄存器 AX 的内容：

```
      ;数据段
      org 100h
varw   dw 1234h,5678h
varb   db 3,4
vard   dd 12345678h
buff   db 10 dup(?)
mess   db 'hello'
      ;代码段
      mov ax,offset mess
      mov ax,type buff+type mess+type vard
      mov ax,sizeof varw+sizeof buff+sizeof mess
      mov ax,lengthof varw+lengthof vard
```

2.14 按照如下输出格式，在屏幕上显示 ASCII 表：

```
      | 0 1 2 3 4 5 6 7 8 9 A B C D E F
-----+-----
20 |  ! " # $ % & ' ( ) * + , - . /
30 |  0 1 2 3 4 5 6 7 8 9 : ; <=> ?
40 |  @ A B C D E F G H I J K L M N O
50 |  P Q R S T U V W X Y Z [ \ ] ^ _
60 |  ` a b c d e f g h i j k l m n o
70 |  p q r s t u v w x y z { | } ~
```

表格最上一行的数字是对应列 ASCII 代码值的低 4 位（用十六进制形式），而表格左边的数字对应行 ASCII 代码值的高 4 位（用十六进制形式）。编程在数据段直接构造这样的表格、填写相应 ASCII 代码值（不是字符本身），然后使用 DOS 的 9 号字符串显示功能实现显示。

2.15 数据段有如下定义，8086 处理器将以小端方式保存在主存：

```
var    dd 12345678h
```

现以字节为单位按地址从低到高的顺序，写出这个变量内容，并说明如下指令的执行结果：

```
mov bx,word ptr var    ;BX=_____
mov cx,word ptr var+2  ;CX=_____
mov dl,byte ptr var    ;DL=_____
mov dh,byte ptr var+3  ;DH=_____
```

2.16 给出 8086 存储器寻址中，有效地址的组成公式，并说明各部分作用。

2.17 说明下列指令中源操作数的寻址方式？假设 VARW 是一个字变量。

- (1) mov dx,1234h
- (2) mov dx,varw
- (3) mov dx,bx
- (4) mov dx,[bx]
- (5) mov dx,[bx+1234h]
- (6) mov dx,varw[bx]
- (7) mov dx,[bx+di]
- (8) mov dx,[bx+si+1234h]

第 3 章 通用数据处理指令

尽管各种处理器支持的指令各不相同，但都有着类同的通用指令。通用指令是处理器的基本指令，主要实现整数处理、程序控制等，本书将在各章逐步介绍。本章介绍 8086 处理器最常用的整数处理指令，即数据传送、算术运算、逻辑运算等基本指令；与此同时，通过阅读简单的程序理解指令功能和工作原理，体会汇编语言的编程方法，并练习编写一些功能简单的程序。

在学习一条指令时，请读者注意如下几个方面：

- ◎ 指令的功能——该指令能够实现何种操作。通常指令助记符就是指令功能的英文单词或其缩写形式。
- ◎ 指令支持的寻址方式——该指令中的操作数可以采用何种寻址方式。前一章最后一节介绍了大多数指令支持的各种寻址方式及其组合，并给出了本书采用的符号。
- ◎ 指令对标志的影响——该指令执行后是否对各个标志位有影响，以及如何影响。
- ◎ 其他方面——例如指令执行时的约定设置、必须预置的参数、隐含使用的寄存器等。

为了更好地理解指令和程序，建议读者一方面可以像前一章那样生成列表文件，静态观察变量分配、指令代码等程序结构（书中不再罗列）；另一方面能够学习调试程序（参见附录 A），动态跟踪指令执行和程序运行，获得更加直观的感受。

3.1 数据传送类指令

数据传送是把数据从一个位置传送到另一个位置，它是计算机中最基本的一种操作。数据传送类指令也是程序设计中最常使用的一种指令。

3.1.1 通用传送指令

最主要的通用传送指令是传送指令 MOV（类似高级语言的赋值语句），有时要使用交换指令 XCHG（类似高级语言的交换函数）。

1. 传送指令 MOV

传送指令 MOV（Move）把一字节或字的操作数从源位置传送至目的位置，可以实现立即数到通用寄存器或到主存的传送，通用寄存器与通用寄存器之间、寄存器与主存或与段寄存器之间，主存与段寄存器之间的传送。

使用前一章最后引入的操作数符号（参见附录 D），MOV 指令的各种组合可以用下列各式表达（斜线“/”表示多种组合形式，注释是说明或功能解释，下同）：

mov reg/mem,imm	;立即数传送
mov reg/mem/seg,reg	;寄存器传送（与段寄存器相互传送只能使用 16 位，下同）
mov reg/seg,mem	;存储器传送
mov reg/mem,seg	;段寄存器传送

在前面的程序中，已经书写了很多 MOV 指令，不再重复。作为练习（参见习题 3.4），

读者不妨为每种操作数组合都列举一个指令实例。这里重点说明一些常见的非法组合。

每一个正确的处理器指令，都对应有指令代码；如果汇编程序无法将你书写的语句翻译成对应的指令代码，那么这个指令就是一条错误的、非法的指令。所以，进行程序设计，首先需要正确书写每一条语句；对常见错误情况，要心中有数。如下几点应该注意：

(1) 8086 指令系统可以对 8 位和 16 位整数类型的数据进行处理，但是双操作数指令（除特别说明）的目的操作数与源操作数的类型必须一致。

例如：

```
MOV SI,DL      ;错误：类型不一致。SI 为 16 位寄存器，DL 为 8 位寄存器
mov si,dx      ;正确：两个 16 位寄存器传送
MOV AL,050AH   ;错误：类型不一致。050AH 超过了寄存器 AL 范围
mov ax,050ah    ;正确：字量数据传送
```

(2) 寄存器名可以表达它的类型属性，变量一经定义也具有类型属性，而立即数和寄存器间接寻址的存储单元等却没有明确的类型。8086 指令系统要求类型一致的两个操作数其中之一必须要有明确的类型，否则用 PTR 指明。

例如：

```
MOV [BX],255   ;错误：无明确类型
mov byte ptr [bx],255 ;正确：BYTE PTR 说明是字节操作
mov word ptr [bx],255 ;正确：WORD PTR 说明是字操作
mov dword ptr [bx],255 ;正确：DWORD PTR 说明是双字操作
```

(3) 为了减小指令编码长度，8086 指令系统没有设计两个存储器操作数的指令（除串操作指令，见 4.3 节），也就不允许两个操作数都是主存单元。

例如：

```
;假设 DBUF1 和 DBUF2 是两个字变量
MOV DBUF2,DBUF1 ;错误：两个操作数都是存储单元
mov ax,dbuf1    ;正确：AX←DBUF1（将 DBUF1 内容送 AX）
mov dbuf2,ax    ;正确：DBUF2←AX（将 AX 内容送 DBUF2）
```

(4) 对专用寄存器可进行操作的指令有限、功能不强，使用时需要注意。

例如：

```
MOV DS,@DATA   ;错误：立即数不能直接传送段寄存器（@DATA 是数据段地址）
mov ax,@data    ;正确：通过 AX 间接传送给 DS
mov ds,ax
```

2. 交换指令 XCHG

交换指令 XCHG（Exchange）用来将 8 位或 16 位源操作数和目的操作数内容交换，可以在通用寄存器与通用寄存器或存储器之间对换数据。使用操作数符号的合法格式如下：

```
xchg reg,reg/mem
xchg reg/mem,reg
```

交换指令的两个操作数实现位置互换，实际上既是源操作数，也是目的操作数。所以无所谓哪个在前、哪个在后，但操作数不能是立即数，也不支持存储器与存储器之间的数据对换。

例如：

```
xchg si,di      ;SI 与 DI 互换内容
xchg si,[di]    ;SI 与 DI 指向的主存单元互换内容
```

8086 处理器采用小端方式存储多字节数据，但有些处理器却采用大端方式。当数据在不

同处理器之间交换时，有时需要进行小端、大端的互换。

例如，双字变量 DVAR 进行小端、大端的互换可以使用交换指令：

```
mov al,byte ptr dvar    ;取第 1 个字节
xchg al,byte ptr dvar+3 ;与第 4 个字节交换
xchg byte ptr dvar,al    ;实现第 1、4 个字节互换（也可以用 MOV 指令）
mov al,byte ptr dvar+1  ;取第 2 个字节
xchg al,byte ptr dvar+2 ;与第 3 个字节交换
xchg al,byte ptr dvar+1 ;实现第 2、3 个字节互换（也可以用 MOV 指令）
```

指令系统中有一条空操作（No Operation）指令，助记符是：NOP。在 8086 处理器中，NOP 指令与指令“XCHG AX,AX”具有同样的指令代码：90H，实际上就是同一条指令。空操作指令看似毫无作用，但处理器执行该指令，需要花费时间，放置在主存中也要占用一字节空间。编程中，使用 NOP 指令可以实现短时间延时，还可以临时占用代码空间以便以后填入需要的指令代码。

3.1.2 堆栈操作指令

处理器通常用硬件支持堆栈（Stack）数据结构。堆栈是一个按照“先进后出”（FILO: First In Last Out）存取原则组织的存储区域，也可以说是“后进先出”（LIFO: Last In First Out）。堆栈具有两种基本操作，对应有两条基本指令：数据压进堆栈操作，对应进栈指令 PUSH；数据弹出堆栈操作，对应出栈指令 POP。

8086 处理器的堆栈建立在主存区域中，使用 SS 段寄存器指向段基地址。堆栈段的范围由堆栈指针寄存器 SP 的初值确定，这个位置就是堆栈底部（不再变化）。堆栈只有一个数据出入口，即当前栈顶（不断变化），由堆栈指针寄存器 SP 的当前值指定栈顶的偏移地址，如图 3-1 所示。随着数据进入堆栈，SP 逐渐减小；数据依次弹出，SP 逐渐增大。随着 SP 增大，弹出的数据不再属于当前堆栈区域；随后进入堆栈的数据也会占用这个存储空间。当然，如果进入堆栈的数据超出了设置的堆栈范围，或者已无数据可以弹出，即 SP 增大到栈底，就会产生堆栈溢出错误。堆栈溢出，轻则使程序出错，重则导致系统崩溃。

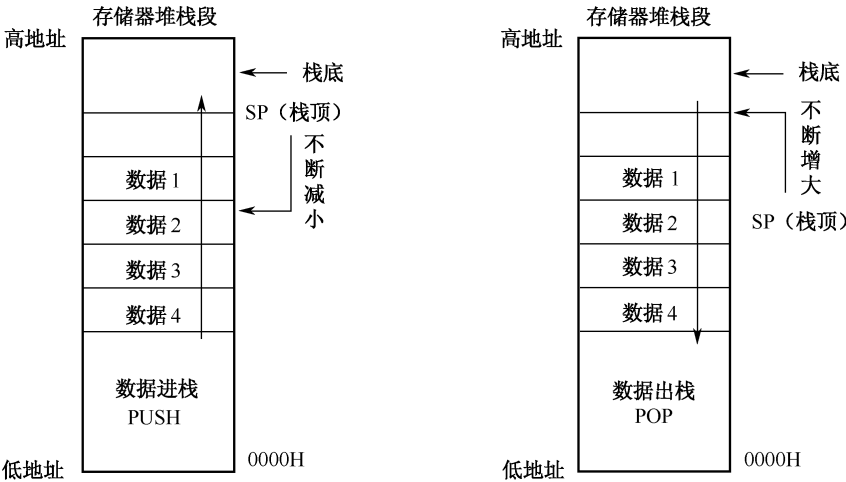


图 3-1 8086 处理器堆栈操作

堆栈操作常被比喻为“摞盘子”。盘子一个压着一个叠起来放进箱子里，就像数据进栈操作；叠起来的盘子应该从上面一个接一个拿走，就像数据出栈操作。最后放上去的盘子，被最先拿走，就是堆栈的“后进先出”操作原则。不过，8086 处理器的堆栈段是“向下生长”的，即随着数据进栈，堆栈顶部（指针 SP）逐渐减小，所以可以想象成为一个倒扣的箱子，盘子（数据）从下面放进去。

1. 进栈指令 PUSH

进栈指令 PUSH 先将 SP 减小作为当前栈顶，然后可以将立即数、通用寄存器和段寄存器内容或存储器操作数传送到当前栈顶。由于目的位置就是栈顶，由 SP 确定，PUSH 指令只表达源操作数。格式是：

push r16/m16/seg ;① SP←SP-2, ② SS:[SP]←r16/m16/seg

8086 处理器的堆栈只能以字为单位操作。进栈字量数据时，SP 向低地址移动 2 字节单元指向当前栈顶，即减 2（像准备了 2 字节的存储单元）；然后数据以“低对低、高对高”的小端方式存放到堆栈顶部，参看图 3-2（a）。

2. 出栈指令 POP

出栈指令 POP 执行与入栈指令相反的功能，它先将栈顶数据传送到通用寄存器、存储单元或段寄存器中，然后 SP 增加作为当前栈顶。由于源操作数在栈顶，由 SP 确定，POP 指令只表达目的操作数。格式是：

pop r16/m16/seg ;① r16/m16/seg←SS:[SP], ② SP←SP+2

出栈字量数据时，首先 SP 向高地址移动 2 字节单元，即加 2；然后数据以“低对低、高对高”原则从栈顶传送目的位置，参看图 3-2（b）。

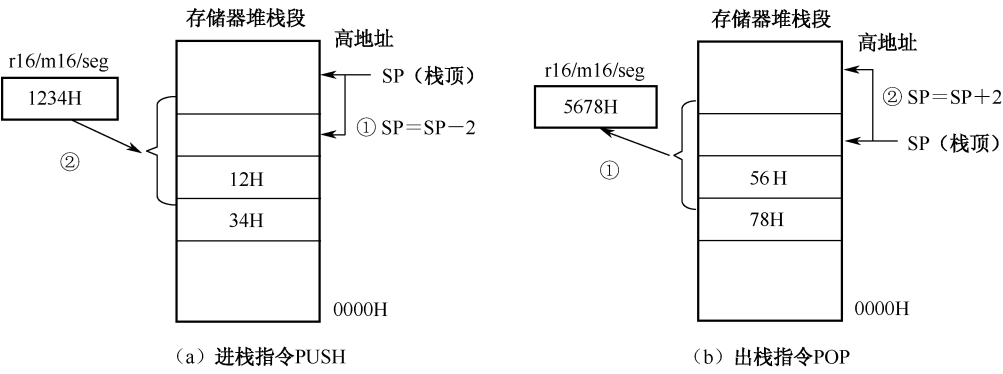


图 3-2 堆栈指令

【例 3-1】堆栈操作程序。

```
      ;数据段
wvar  dw 1234h,5678h
      ;代码段
mov ax,wvar      ;AX←WVAR 变量值 1234H
push ax          ;将 AX 内容压入堆栈
push wvar+2      ;将变量 WVAR 第 2 个数据压入堆栈
pop ax           ;栈顶数据弹出到 AX
pop wvar         ;栈顶数据弹出到 WVAR 位置
```

2 条 PUSH 指令分别压入堆栈的数据是 1234H 和 5678H，每次 SP 减 2。按照“后进先出”原则，接着的 2 条 POP 指令执行结果是：AX=5678H，WVAR 变量第一个字数据位置还是当初压入的内容 1234H。

想要更直观体会堆栈操作，可以利用调试程序。建议读者先通过附录 A 调试程序 DEBUG，了解其常用命令的使用，然后参考前一章 2 个使用示例，熟悉汇编和执行单条指令的方法，本章则掌握汇编和执行程序片段的过程。

图 3-3 和图 3-4 演示例 3-1 堆栈操作程序片段的调试过程，操作步骤详述如下：

```

-a
0D59:0100 mov ax,[2000]
0D59:0103 push ax
0D59:0104 push [2002]
0D59:0106 pop ax
0D59:0109 pop [2000]
-a 2000
0D59:2000 dw 1234,5678
0D59:2004
-d 2000 200f
0D59:2000 34 12 78 56 70 65 63 69-66 79 20 61 20 73 69 6E 4.xUpecify a sin
-t
AX=1234 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0103 NU UP EI PL NZ NA PO NC
0D59:0103 50 PUSH AX

```

图 3-3 堆栈操作的调试截图 1

```

0D59:0103 50 PUSH AX
-t 2
AX=1234 BX=0000 CX=0000 DX=0000 SP=FFEC BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0104 NU UP EI PL NZ NA PO NC
0D59:0104 FF360220 PUSH [2002]
DS:2002=5678
AX=1234 BX=0000 CX=0000 DX=0000 SP=FFEA BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0108 NU UP EI PL NZ NA PO NC
0D59:0108 58 POP AX
-d ss:ffea
0D59:FFEB 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....xU4...
0D59:FFFB 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....
-t 2
AX=5678 BX=0000 CX=0000 DX=0000 SP=FFEC BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0109 NU UP EI PL NZ NA PO NC
0D59:0109 8F060020 POP [2000]
DS:2000=1234
AX=5678 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=010D NU UP EI PL NZ NA PO NC
0D59:010D 71 XCHG CX,AX

```

图 3-4 堆栈操作的调试截图 2

(1) 在 DOS 命令行提示符输入 DEBUG，进入调试程序。

(2) 在调试程序提示符下输入汇编命令 A，进入汇编状态。

(3) 如图 3-3 输入例 3-1 代码段的指令，其中变量 WVAR 可以使用[2000]，则 WVAR+2 是[2002]。变量 WVAR 的地址当然还可以选择其他值，但是注意不要与从 0100H 开始的指令代码冲突，也不要安排在段的末端占用堆栈空间（因为 DEBUG 调试程序默认 4 个段从同一个地址起始）。此时还可以使用反汇编命令 U 查看刚才输入的指令，确认正确，以防出错。

(4) 可以使用修改命令 E 保存 WVAR 变量值。本例采用汇编的方法，因为 DEBUG 的汇编状态支持 DB 和 DW 伪指令。所以，使用汇编命令“A 2000”再次进入汇编状态，但指定了变量所在的地址（A 命令没有地址参数，会接着上一条指令位置）。接着，输入“DW 1234,5678”保存 2 个 16 位数据，占 4 字节。

(5) 显示命令 D (Dump) 用于显示存储单元内保存的内容。通常 D 命令后面要指明存储单元地址，默认以字节为单位显示 8 行、每行 16 个存储单元的内容（在 DOS 默认的 80 列显示模式下）。显示命令在屏幕上先显示出存储单元的逻辑地址，接着以十六进制形式显示每个字节的数值，一行 16 字节，中间用短线分隔前 8 个和后 8 个，最后面则是将内容以 ASCII 值显示对应的字符，对不可显示的 ASCII 码显示为点“.”。例如，数值“34”的内容，由于等于数码“4”的 ASCII 值，所以后面显示字符“4”。再如，下一个字节数值是“12”，是 ASCII 码的控制字符，不可显示，后面以点表示。本例中只有 4 字节内容，所以还显示出

结束的地址 2000F，这样恰好显示一行 16 字节。

(6) 确认输入的指令和数据无误，就可以开始执行每条指令。使用跟踪命令 T 执行第一条指令后暂停，通过显示结果观察到 AX=1234H。注意此时的堆栈指针 SP=FFEEH。

(7) 接着如图 3-4 执行堆栈操作指令，在跟踪命令 T 后加一个数值 2，表示单步执行 2 条指令，不过每条指令执行后都会显示结果，以便观察。可以看到，每执行一条 PUSH 指令，SP 减少 2，SP 从 FFEEH 减 2 为 FFECH，再减 2 为 FFEAH，此为当前堆栈顶部。

(8) 有 2 个数据压入堆栈，可以使用显示命令从当前栈顶位置观察当前堆栈里的内容。显示堆栈段内容，最好在 D 命令后使用堆栈段寄存器前缀“SS:”，然后再给出当前栈顶偏移地址，以免误显示其他段的内容。此时的 D 命令虽然没有限制显示范围，但已经显示到了该段尾部，所以不再继续显示下一个段内容。

(9) 继续单步执行 2 条堆栈弹出指令，每执行一条 POP 指令，SP 加 2，又恢复为最初的 FFEEH，实现了堆栈平衡。还可以注意到第一条 POP 指令使得 AX 得到 WVAR+2 的内容 5678H，也就是“PUSH WVAR+2”和“POP AX”借用堆栈实现了指令“MOV AX,WVAR+2”的功能 (AX←WVAR+2)。第 2 条 POP 指令，配合第一条 PUSH 指令仍然让 WVAR 变量保持原值未变，在实际应用中相当于恢复原值。读者可以继续使用 D 命令查看数据。

3. 堆栈的应用

堆栈是程序中不可或缺的一个存储区域。除堆栈操作指令外，还有子程序调用 CALL 和子程序返回 RET、中断调用 INT 和中断返回 IRET 等指令，以及内部异常、外部中断等情况都会使用堆栈、修改 SP 值（将在后续章节中逐渐展开）。

堆栈可用来临时存放数据，以便随时恢复它们。使用 POP 指令时，应该明确当前栈顶的数据是什么，可以按程序执行顺序向前观察由哪个操作压入了该数据。

既然堆栈是利用主存实现的，我们当然还能以随机存取方式读写其中的数据。通用寄存器之一的堆栈基址指针 BP 就是出于这个目的而设计的。

例如：

```
mov bp,sp           ;BP←SP
mov ax,[bp+4]        ;AX←SS:[BP+4], BP 默认与堆栈段配合
mov [bp],ax          ;SS:[BP]←AX
```

利用堆栈实现主、子程序间传递参数就利用上述方法，这也是堆栈的主要作用之一。

堆栈还常用于子程序的寄存器保护和恢复。由于堆栈的栈顶和内容随着程序的执行不断变化，所以编程时要注意入栈和出栈的数据要成对，才能保持堆栈平衡。

3.1.3 其他传送指令

指令系统中还有一些针对特定需要设计的专用传送指令。

1. 地址传送指令

存储器操作数具有地址属性，地址传送指令可以获取其地址。其中，最常用的是获取有效地址指令 LEA (Load Effective Address)，格式如下：

```
lea r16,mem          ;r16←mem 的有效地址 EA (不需类型一致)
```

LEA 指令将存储器操作数的有效地址（段内偏移地址）传送到 16 位通用寄存器中。它的作用等同于汇编程序 MASM 的地址操作符 OFFSET。但是，LEA 指令是在指令执行时计

算出偏移地址，OFFSET 操作符是在汇编阶段取得变量的偏移地址，后者执行速度更快。不过，对于在汇编阶段无法确定的偏移地址，就只能利用 LEA 指令获取了。

【例 3-2】 地址传送程序。

```

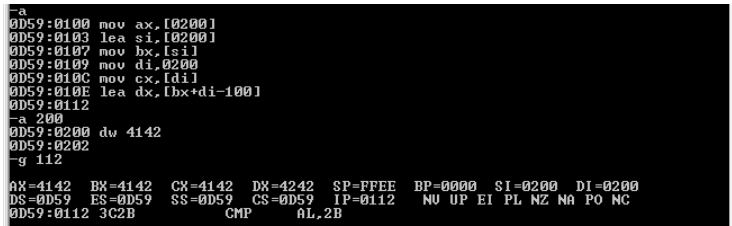
;数据段
wvar    dw 4142h
;代码段
mov ax,wvar      ;直接寻址获得变量值: AX=4142H
lea si,wvar      ;执行时获得变量地址: SI 指向 WVAR
mov bx,[si]      ;通过地址获得变量值: BX=4142H
mov di,offset wvar ;汇编时获得变量地址: DI 指向 WVAR
mov cx,[di]      ;通过地址获得变量值: CX=4142H
lea dx,[bx+di-100h] ;实现运算功能: DX=BX+DI-100H
```

前一条 LEA 指令使 SI 等于 WVAR 变量的有效地址（并没有读取变量内容），与利用 OFFSET 设置的 DI 相同，所以 AX=BX=CX=4142H。然而，利用 OFFSET 操作符在源程序汇编时已经计算出地址，实际上是一个立即数。

后一条 LEA 指令实际上先进行加法运算得到偏移地址（相对基址变址寻址方式），然后传送给 DX 寄存器。有时也利用 LEA 指令的这个特点实现加法运算。然而，这条指令不能使用“mov dx,offset [bx+di-100h]”语句替代，因为汇编时并不知道执行时 BX 和 DI 等于什么，它根本就是非法的。

8086 处理器还有指针传送指令 LDS 和 LES，它们能将主存连续 4 字节内容的前两个分别传送给 DS 和 ES，后续字节作为偏移地址传送给指令的 16 位通用寄存器。

图 3-5 演示例 3-2 地址传送程序片段的调试过程，操作步骤简述如下：



```

-a
0D59:0100 mov ax,[0200]
0D59:0103 lea si,[0200]
0D59:0107 mov bx,[si]
0D59:0109 mov di,0200
0D59:010C mov cx,[di]
0D59:010E lea dx,[bx+di-100]
0D59:0112
-a 200
0D59:0200 dw 4142
0D59:0202
-g 112
AX=4142 BX=4142 CX=4142 DX=4242 SP=FFEE BP=0000 SI=0200 DI=0200
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0112 NU UP EI PL NZ NA PO NC
0D59:0112 3C2B CMP AL,2B
```

图 3-5 地址传送的调试截图

（1）在调试程序的汇编状态，输入例 3-2 代码段指令和数据段变量值。注意，本例将变量安排在 0200H（读者也可以设置在其他位置）。

（2）由于各条指令的执行结果之间没有关联，不必一条一条指令的单步执行，可以使用运行命令 G（Go）连续将这 6 条指令执行完毕，再观察结果。使用运行命令通常需要设置断点，也就是停止执行指令的地址，本例所设置的断点位置是偏移地址 0112H，所以使用“G 112”命令。

2. 换码指令

数据表是常见的数据结构，编程中经常需要获得数据表中某个特定的数据项，处理器为此专门设计了换码指令。换码指令 XLAT（Translate）是一条比较复杂的指令，但格式却非常简单，如下所示：

xlata;AL←[BX+AL]

使用 XLAT 指令前，需要将 BX 指向主存缓冲区（即数据表首地址），给 AL 赋值距离缓

缓冲区开始的位移量（即表中数据项的位置），执行的功能是将缓冲区该位移量位置的数据取出赋给 AL，可以表达为“AL←[BX+AL]”。由于 XLAT 指令隐含使用 BX 和 AL，所以其助记符后无须写出操作数，默认该缓冲区在 DS 数据段；如果设置的缓冲区在其他段，则需要写明缓冲区的变量名，汇编程序就会加上必要的段超越前缀，用户也可以在变量名前加上段超越前缀。

【例 3-3】 换码显示程序。

前面通过多次使用 9 号 DOS 功能，掌握了字符串显示的方法。现在引出显示一个字符的方法——使用 2 号 DOS 功能（本书配套的输入输出子程序库的 DISPC 子程序也同样可以实现一个字符显示），这个方法要求（参见附录 B）：

（1）设置 DL 等于要显示字符的 ASCII 码，例如显示问号的指令：

```
mov dl,'?'
```

（2）赋值 AH 等于 2，即 2 号功能，指令可以是：

```
mov ah,2
```

（3）实现调用，即将字符显示在当前光标处，指令是：

```
int 21h
```

现在利用 DOS 的 2 号功能，实现主存中“0~9”之间数字的显示。为此，按顺序设置一个字符“0”~“9”的 ASCII 表，取名 TAB，利用换码指令将某个数字转换成对应的 ASCII 就可以显示了，程序如下：

```
                ;数据段
num             db 6,7,7,8,3,0,0,0      ;要被转换的数字
tab             db '0123456789'          ;代码表
                ;代码段
                mov cx,lengthof num       ;CX 等于要转换数字的个数
                mov si,offset num         ;SI 指向要转换的数字
                mov bx,offset tab         ;BX 指向代码表
again:          mov al,[si]               ;AL←要转换的数字
                xlat                       ;换码，AL 得到数字的 ASCII 值
                mov dl,al                  ;显示
                mov ah,2
                int 21h
                add si,1                   ;指向下一个数字
                loop again                 ;循环
```

代码表建立后，要将表格的首地址存放于 BX 寄存器，需要转换的数字存放于 AL 寄存器，并应等于被转换的代码在相对表格首地址的位移量。设置好后，执行换码指令，即将 AL 寄存器的内容转换为目的代码。为了实现多个数字转换，程序中使用了将在下章学习的循环指令 LOOP。LOOP 指令的功能是，先将本指令默认使用的计数器寄存器 CX 减 1，然后判断 CX 是否为 0：如果 CX 不等于 0，说明循环没有结束，则程序跳转到 LOOP 指令所指定的标号位置执行那里的指令；如果 CX 等于 0，说明循环结束，程序按顺序向下执行。这里实现了 AGAIN 标号与 LOOP 指令之间的指令重复执行 CX 设定的次数，即要转换数字的个数。本示例程序执行结束，显示数字：67783000。

XLAT 是一种具有特定功能的指令，常用于将一种代码转换为另一种代码，但只能进行字节量表换码，AL 作位移量又限制其最大为 255。如果不存在该指令，可以用下段程序完成同样功能：

```

;代码段（未使用 XLAT，但按照其功能编程）
mov cx,lengthof num
mov si,offset num
again:  mov bx,offset tab
        mov al,[si]           ;AL←要转换的数字
        mov ah,0              ;AH←0（下节有解释）
        add bx,ax              ;BX←BX+AX，指向对应的字符
        mov al,[bx]           ;换码
        mov dl,al              ;显示
        mov ah,2
        int 21h
        add si,1               ;指向下一个数字
        loop again             ;循环

```

既然不使用 XLAT 指令，就没有必要按其功能编程。例如，利用寄存器相对寻址具有的计算能力，也可以实现换码：

```

;代码段（未使用 XLAT，利用寄存器相对寻址编程）
mov cx,lengthof num
mov si,offset num
again:  mov bl,[si]           ;BL←要转换的数字
        mov bh,0              ;BH←0
        mov dl,tab[bx]        ;换码
        mov ah,2              ;显示
        int 21h
        add si,1               ;指向下一个数字
        loop again             ;循环

```

3. 标志传送指令

8086 处理器有可以直接改变标志 CF、DF、IF 状态的标志位操作指令，还有针对标志寄存器低 8 位和全部 16 位传送的指令，如表 3-1 所示。

表 3-1 标志位操作指令

指 令	功 能
cld	复位进位标志：CF=0
std	置位进位标志：CF=1
cmc	求反进位标志：原为 0 变为 1，原为 1 变为 0
cld	复位方向标志：DF=0，串操作后地址增大
std	置位方向标志：DF=1，串操作后地址减小
cli	复位中断标志：IF=0，禁止可屏蔽中断
sti	置位中断标志：IF=1，允许可屏蔽中断
lahf	标志寄存器低字节内容传送到 AH 寄存器
sahf	AH 寄存器内容传送到标志寄存器低字节
pushf	标志寄存器压入堆栈
popf	堆栈顶部一个字量数据弹出到标志寄存器

尽管许多指令的执行都会影响标志，但这组指令能够直接操作标志寄存器。当有必要了解当前标志状态或设置标志状态时可以使用这组指令。

4. 输入输出指令

计算机外部设备需要通过输入输出接口电路（简称 I/O 接口）与主机相连。对外设编程实际上是针对 I/O 接口电路编程。I/O 接口电路呈现给程序员的则是各种可编程寄存器，计算机系统使用编号区别各个 I/O 接口寄存器，这就是输入输出地址或 I/O 地址（与存储器地址相互独立），也常用更形象化的术语：I/O 端口（Port）。通过访问 I/O 端口进行外部操作要使用输入输出指令，或简称 I/O 指令，它也属于基本的数据传送类指令。

8086 处理器的常用指令都可以存取存储器操作数，但存取 I/O 端口实现输入输出的指令很少。简单地说，只有两种：输入指令 IN 和输出指令 OUT。

助记符 IN 表示输入指令，实现数据从 I/O 接口输入到处理器，格式如下：

```
in al/ax, i8/dx
```

助记符 OUT 表示输出指令，实现数据从处理器输出到 I/O 接口，格式如下：

```
out i8/dx, al/ax
```

(1) I/O 寻址方式

8086 处理器有多种存储器寻址方式可以访问存储单元。但是，访问 I/O 接口时只有两种寻址方式：直接寻址和 DX 间接寻址。

I/O 地址的直接寻址，由 I/O 指令直接提供 8 位 I/O 地址，只能寻址最低 256 个 I/O 地址（00~FFH）。格式中用 i8 表示这个直接寻址的 8 位 I/O 地址。虽然形式上与立即数一样，但它们被应用于 IN 或 OUT 指令就表示直接寻址的 I/O 地址。

I/O 地址的间接寻址，用 DX 寄存器保存访问的 I/O 地址。由于 DX 是 16 位寄存器，所以可寻址全部 I/O 地址（0000~FFFFH）。I/O 指令中直接书写成 DX，表示 I/O 地址。

8086 处理器的 I/O 地址共 64K 个（0000~FFFFH），每个地址对应一个 8 位端口，不需要分段管理。最低 256 个（00~FFH）可以用直接寻址或间接寻址访问，高于 256 的 I/O 地址只能使用 DX 间接寻址访问。

(2) I/O 数据传输量

IN 和 OUT 指令只允许通过累加器 AX 与外设交换数据。8 位 I/O 指令使用 AL，16 位 I/O 指令使用 AX。执行输入指令 IN，外设数据进入处理器的 AL/AX 寄存器（作为目的操作数，被书写在左边）。执行 OUT 输出指令，处理器数据通过 AL/AX 送出去（作为源操作数，被写在右边）。

例如：

```
in al,21h           ;从地址为 21H 的 I/O 端口读一个字节数据到 AL
mov dx,300h         ;DX 指向 300H 端口
out dx,al           ;将 AL 中的字节数据送到地址为 300H (DX) 的 I/O 端口
```

两种 I/O 寻址和两种数据传输量的各种组合参见表 3-2。

表 3-2 输入输出指令示例

输入指令 IN	输出指令 OUT
in al,20h	out 20h,al
in ax,20h	out 20h,ax
mov dx,3fch	mov dx,3fch
in al,dx	out dx,al
in ax,dx	out dx,ax

16 位 80x86 处理器只支持使用 AL 和 AX 的 8 位和 16 位输入输出指令。类似字节寻址的存储单元，每个 I/O 地址也是对应一个 8 位外设端口，进行字量输入输出，实际上是从连续的两个端口输入输出 2 字节，原则是“低地址对应低字节数据、高地址对应高字节数据”的小端方式。

例如：

```
in al,61h           ;从 I/O 地址 61H 读 1 字节数据到 AL
mov ah,al           ;转存到 AH
in al,60h           ;从 I/O 地址 60H 读 1 字节数据到 AL
```

如果有电路支持，上述程序片段可以使用如下指令替代，同样实现从 60H 和 61H 端口读取一个字到 AX 的功能：

```
in ax,60h
```

3.2 算术运算类指令

算术运算是数据对数据进行加减乘除，是基本的数据处理方法。加减运算除有“和”或“差”的结果外，还有进借位、溢出等状态标志；状态标志也是结果的一部分。所以，本小节首先讨论处理器的各种状态标志，然后再学习算术运算指令。

3.2.1 状态标志

3.1 节介绍的数据传送类指令中，除了标志为目的操作数的标志传送指令外，其他传送指令并不影响标志；也就是说，标志并不因为传送指令的执行而改变，所以我们并没有涉及标志问题。但现在需要了解它们了。

状态标志一方面作为加减运算和逻辑运算等指令的辅助结果，另一方面又用于构成各种条件、实现程序分支，是汇编语言编程中非常重要的一个方面。

8086 的标志寄存器及 9 个标志参见图 1-5，本节讨论其中 6 个状态标志。

1. 进位标志 CF

处理器设计的进（借）位 CF（Carry Flag）标志类似十进制数据加减运算中的进位和借位，不过只是体现二进制数据最高位的进位或借位。具体来说，当加减运算结果的最高位有进位（加法）或借位（减法）时，将设置进位标志为 1，即 $CF=1$ ；如果没有进位或借位，则设置进位标志为 0，即 $CF=0$ 。换句话说，加减运算后，如果 $CF=1$ ，说明数据运算过程中出现了进位或借位；如果 $CF=0$ ，说明没有进位或借位。

例如，有两个 8 位二进制数：00111010 和 01111100。如果将它们相加，运算结果是：10110110。运算过程中，最高位没有向上再进位，所以这个运算结果将使得 $CF=0$ 。但如果是 10101010 和 01111100 相加，结果是[1]00100110，出现了向高位进位（用了中括号表达），那么这个运算结果将使得 $CF=1$ 。

进位标志是针对无符号整数运算设计的，反映无符号数据加减运算结果是否超出范围、是否需要利用进（借）位反映正确结果。 N 位无符号整数表达的范围是 $0 \sim 2^N - 1$ 。如果相应位数的加减运算结果超出了其能够表达的这个范围，即产生了进位或借位。

上面例子中二进制数 $00111010 + 01111100 = 10110110$ ，将它们转换成十进制表达是 $58 +$

$124=182$ 。运算结果 182 仍在 $0\sim 255$ 范围之内，没有产生进位，所以 $CF=0$ 。

对于二进制数 $10101010+01111100=[1]00100110$ ，将它们转换成十进制表达是 $170+124=294=256+38$ 。运算结果 294 超出了 $0\sim 255$ 范围，所以将使得 $CF=1$ 。这里，进位 $CF=1$ 表达了十进制数据 256。

2. 溢出标志 OF

把水倒入茶杯时，如果倒入了超出茶杯容量的水，水会漫出来，这就是溢出的本意：一个容器不能存放超过其容积的物体。同样道理，处理器设计的溢出标志 OF (Overflow Flag) 用于表达有符号整数进行加减运算的结果是否超出范围。超出范围，就是有溢出，将设置溢出标志 $OF=1$ ；没有溢出，则 $OF=0$ 。

溢出标志是针对有符号整数运算设计的，反映有符号数据加减运算结果是否超出范围。处理器默认采用补码形式表示有符号整数， N 位补码表达的范围是 $-2^{N-1}\sim +2^{N-1}-1$ 。如果有符号整数运算结果超出了这个范围，就是产生了溢出。

对上面例子的两个 8 位二进制数 00111010 和 01111100 ，按照有符号数的补码规则它们都是正整数，用十进制表达分别是：58 和 124。它们求和的结果是二进制 10110110 ，十进制表达 $58+124=182$ 。运算结果 182 超出了 $-128\sim +127$ 范围，产生溢出，所以 $OF=1$ 。另一方面，按照补码规则，8 位二进制结果 10110110 ，最高位为 1 实际上表达的是负数，所以溢出情况下的运算结果是错误的。

对于二进制数 10101010 ，最高位是 1，按照补码规则表达负数，求反加 1 得到绝对值，即表达十进制数：-86。它与二进制数 01111100 (十进制表达 124) 相加，结果是： $[1]00100110$ 。因为进行有符号数据运算，所以不考虑无符号运算出现的进位， 00100110 才是我们需要的结果：38 ($=-86+124$)。运算结果 38 没有超出 $-128\sim +127$ 范围，将使得 $OF=0$ 。所以，有符号数据进行加减运算，只有在没有溢出情况下才是正确的。

注意，溢出标志 OF 和进位标志 CF 是两个意义不同的标志。进位标志表示无符号整数运算结果是否超出范围，超出范围后加上进位或借位运算结果仍然正确；而溢出标志表示有符号整数运算结果是否超出范围，超出范围运算结果不正确。处理器对两个操作数进行运算时，按照无符号整数求得结果，并相应设置进位标志 CF；同时，根据是否超出有符号整数的范围设置溢出标志 OF。应该利用哪个标志，则由程序员来决定。也就是说，如果将参加运算的操作数认为是无符号数，就应该关心进位；认为是有符号数，则要注意是否溢出。

处理器利用异或门等电路判断运算结果是否溢出。按照处理器硬件的方法或者前面论述的原则进行判断会比较麻烦，这里给出一个简单规则：只有当两个相同符号数相加（含两个不同符号数相减），而运算结果的符号与原数据符号相反时，产生溢出；因为，此时的运算结果显然不正确。其他情况下，则不会产生溢出。

3. 其他状态标志

零标志 ZF (Zero Flag) 反映运算结果是否为 0。运算结果为 0，则设置 $ZF=1$ ，否则 $ZF=0$ 。例如 8 位二进制数 $00111010+01111100=10110110$ ，结果不是 0，所以设置 $ZF=0$ 。如果是 8 位二进制数 $10000100+01111100=[1]00000000$ ，最高位进位有进位 CF 标志反映，除此之外的结果是 0，所以这个运算结果将使得 $ZF=1$ 。注意，零标志 $ZF=1$ ，反映结果是 0。

符号标志 SF (Sign Flag) 反映运算结果是正数还是负数？其是通过符号位可以判断数据

的正负，因为符号位是二进制数的最高位，所以运算结果最高位（符号位）就是符号标志的状态。即运算结果最高位为 1，则 SF=1；否则 SF=0。例如 8 位二进制数 00111010+01111100=10110110，结果最高位是 1，所以设置 SF=1。如果是 8 位二进制数 10000100+01111100=[1]00000000，最高位是 0（进位 1 不是最高位），所以这个运算结果将使得 SF=0。

奇偶标志 PF（Parity Flag）反映运算结果最低字节中“1”的个数是偶数还是奇数，便于用软件编程实现奇偶校验。最低字节中“1”的个数为零或偶数时，PF=1；为奇数时，PF=0。例如 8 位二进制数 00111010+01111100=10110110，结果中“1”的个数为 5 个，是奇数，故设置 PF=0。如果是 8 位二进制数 10000100+01111100=[1]00000000，除进位外的结果是零个“1”，所以这个运算结果将使得 PF=1。注意，PF 标志仅反映最低 8 位中“1”的个数是偶或奇，就是进行 16 位或 32 位操作也是这样。

加减运算结果将同时影响上述 5 个标志，表 3-3 总结了前面示例，便于对比理解。

表 3-3 加法运算结果对标志的影响

加法运算及其结果	CF	OF	ZF	SF	PF
00111010+01111100=[0] 10110110	0	1	0	1	0
10101010+01111100=[1] 00100110	1	0	0	0	0
10000100+01111100=[1] 00000000	1	0	1	0	1

辅助进位标志 AF（Auxiliary carry Flag），现在称为调整标志 AF（Adjust Flag），反映加减运算时最低半字节有无进位或借位。最低半字节（即 D3 位向 D4 位）有进位或借位时，AF=1；否则 AF=0。这个标志主要由处理器内部使用，用于十进制算术运算的调整指令，用户一般不必关心。例如 8 位二进制数 00111010+01111100=10110110，低 4 位有进位，所以 AF=1。

3.2.2 加法指令

加法指令主要包括 ADD、ADC 和 INC 指令，除 INC 不影响进位标志 CF 外，其他指令按定义影响全部状态标志位，即按照运算结果相应设置各个状态标志为 0 或 1。

1. 加法指令 ADD

加法指令 ADD 将目的操作数与源操作数相加，结果送到目的操作数，格式如下：

add reg,imm/reg/mem ;加法: reg←reg+imm/reg/mem

add mem,imm/reg ;加法: mem←mem+imm/reg

它支持寄存器与立即数、寄存器、存储单元，以及存储单元与立即数、寄存器间的加法运算，按照定义影响 6 个状态标志位。

例如：

```

mov ax,9348h      ;AX←9348H，不影响标志
add al,47h        ;AL←AL+47H=48H+47H=8FH，所以 AX=938FH
                  ;状态标志：OF=1，SF=1，ZF=0，PF=0，CF=0
add ax,6ffh       ;AX←AX+6FFFH=938FH+6FFFH=038EH
                  ;状态标志：OF=0，SF=0，ZF=0，PF=1，CF=1

```

算术运算类既可以进行 8 位运算，也可以进行 16 位运算。对于 8 位运算指令，状态标志反映 8 位运算结果的状态；同样，进行 16 位运算，状态标志（除 PF）反映 16 位运算结果

的状态。例如进位 CF 标志在进行 8 位加法时反映最高位 D7 的向上进位，而进行 16 位加法则反映最高位 D15 的进位。

为了查看指令对状态标志的影响情况，可以进入调试程序进行单步执行（如图 3-6 所示，其中 NV、OV 等符号含义可以参考附录 A），也可以调用本书提供的显示状态标志的子程序 DISPRF（参见附录 C 输入输出子程序库）。作为练习，大家可以将这个调用语句加入上述每条指令后形成一个源程序，生成可执行文件运行，对比显示结果。

```

-a
0D59:0100 mov ax,9348
0D59:0103 add al,47
0D59:0105 add ax,6fff
0D59:0108
-t 3
AX=9348 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0103  NO UP EI PL NZ NA PO NC
0D59:0103 0447      ADD     AL,47
AX=938F BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0105  NO UP EI NG NZ NA PO NC
0D59:0105 05FF6F    ADD     AX,6FFF
AX=038E BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0108  NO UP EI PL NZ AC PE CY
0D59:0108 1CBF      SBB     AL,BF

```

图 3-6 加法指令的调试截图

2. 带进位加法指令 ADC

带进位加法指令 ADC（Add with Carry）除完成 ADD 加法运算外，还要加上进位 CF，结果送到目的操作数，按照定义影响 6 个状态标志位，格式如下：

adc reg,imm/reg/mem ;带进位加法：reg←reg+imm/reg/mem+CF

adc mem,imm/reg ;带进位加法：mem←mem+imm/reg+CF

ADC 指令的设计目的是用于与 ADD 指令相结合实现多精度数的加法。8086 处理器可以直接用一条实现 16 位数据的加法。但是，多于 16 位的数据相加就需要先将两个操作数的低 16 位相加（用 ADD 指令），然后再加高位部分并将进位加到高位（需要用 ADC 指令）。

【例 3-4】 32 位数据相加程序。

```

;数据段
dvar1 dd 82347856h ;32 位数据 1
dvar2 dd 12348998h ;32 位数据 2
;代码段
mov ax,word ptr dvar1 ;取数据 1 的低 16 位
add ax,word ptr dvar2 ;加数据 2 的低 16 位，设置 CF
mov dx,word ptr dvar1+2 ;取数据 1 的高 16 位
adc dx,word ptr dvar2+2 ;加数据 2 的高 16 位，同时要加上 CF

```

本例程序实现两个 32 位整数相加，和值保存在 DX（高 16 位）和 AX（低 16 位）寄存器中。32 位数据用双字变量定义 DD，先加低 16 位（ADD 指令），再加高 16 位（ADC 指令）。进行高 16 位加法时，需要加上低 16 位相加形成的进位标志 CF，所以使用了带进位的加法指令。MOV 指令不影响任何状态标志，所以执行 ADC 指令时使用的 CF 就是前面 ADD 指令设置的状态。与 16 位寄存器配合，具有 32 位属性的变量需要进行强制类型转换，用 WORD PTR DVAR1 指向低 16 位，用 WORD PTR DVAR1+2 指向高 16 位。

3. 增量指令 INC

增量指令 INC（Increment）只有一个操作数，对操作数加 1（增量）再将结果返回原处。操作数可以是寄存器或存储单元，格式如下：

inc reg/mem ;加 1：reg/mem←reg/mem+1

设计增量指令的目的，主要用于对计数器和地址指针的调整，所以它不影响进位 CF 标志，但影响其他状态标志位。

例如：

```
inc cx           ;字量数据加 1: CX←CX+1
inc byte ptr [bx] ;字节量数据加 1: [BX]←[BX]+1
```

3.2.3 减法指令

减法指令主要包括 SUB、SBB、DEC、NEG 和 CMP，除 DEC 不影响 CF 标志外，其他按定义影响全部状态标志位。

1. 减法指令 SUB

减法指令 SUB（Subtract）使目的操作数减去源操作数，结果送到目的操作数，格式如下：

```
sub reg,imm/reg/mem ;减法: reg←reg-imm/reg/mem
sub mem,imm/reg      ;减法: mem←mem-imm/reg
```

像 ADD 指令一样，SUB 指令支持寄存器与立即数、寄存器、存储单元，以及存储单元与立即数间的减法运算，按照定义影响 6 个状态标志位。

例如：

```
mov ax,9348h      ;AX←9348H
sub al,47h         ;AX←9301H, OF=0, SF=0, ZF=0, PF=0, CF=0
sub ax,6ffh        ;AX←2302H, OF=1, SF=0, ZF=0, PF=0, CF=0
```

2. 带借位减法指令 SBB

带借位减法指令 SBB（SuBtract with Borrow）除完成 SUB 减法运算外，还要减去借位 CF，结果送到目的操作数，按照定义影响 6 个状态标志位，格式如下：

```
sbb reg,imm/reg/mem ;减法: reg←reg-imm/reg/mem-CF
sbb mem,imm/reg      ;减法: mem←mem-imm/reg-CF
```

SBB 指令主要用于与 SUB 指令相结合实现多精度数的减法。多于 16 位数据的减法需要先将两个操作数的低 16 位相减（用 SUB 指令），然后再减高位部分，并从高位减去借位（需要用 SBB 指令）。

3. 减量指令 DEC

减量指令 DEC（Decrement）对操作数减 1（减量）再将结果返回原处，格式如下：

```
dec reg/mem      ;减 1: reg/mem←reg/mem-1
```

DEC 指令对应 INC 指令，也主要用于对计数器和地址指针的调整，不影响进位 CF 标志，但影响其他状态标志位。

例如：

```
dec cl           ;字节量数据减 1: CL←CL-1
dec word ptr [bx] ;字量数据减 1: [BX]←[BX]-1
```

【例 3-5】大小写字母转换程序。

```
        ;数据段
msg     db 'welcome','$'
        ;代码段
```



```

        mov cx,(lengthof msg)-1    ;CX 等于字符串长度
        mov bx,0                    ;BX←0, 指向头一个字母
again:   sub msg[bx],'a'-'A'        ;小写字母减 20H 转换为大写
        inc bx                      ;指向下一个字母
        loop again                  ;循环
        mov dx,offset msg
        mov ah,9
        int 21h                    ;显示

```

编程中经常需要对英文字母大小写进行转换。本例程序将小写字母组成的字符串改为大写字母，然后显示。小写和对应的大写字母相差 20H (= 'a' - 'A' = 61H - 41H)，所以小写字母减 20H 成为大写字母，反过来大写字母加 20H 就成为小写字母。给定的字符串全部由小写组成，所以程序没有判断是否是小写字母(判断方法在下章介绍)。本例程序的减法指令用 MSG[EBX] 指向字符串中的字母，是寄存器相对寻址的目的操作数，MSG 表示字符串首位置，BX 指向字符串中的字母。执行过程中先取出小写字母减 20H 后成为大写字母，又保存到原来位置。

4. 求补指令 NEG

求补指令 NEG (Negative) 也是一个单操作数指令，它对操作数执行求补运算，即用零减去操作数，然后结果返回操作数。

```
neg reg/mem    ;用 0 作减法: reg/mem=0-reg/mem
```

NEG 指令对标志的影响与用零作减法的 SUB 指令一样，可用于对负数求补码或由负数的补码求其绝对值。

例如：

```

mov ax,0ff64h
neg al          ;AX=FF9CH, OF=0, SF=1, ZF=0, PF=1, CF=1
sub al,9dh      ;AX=FFFFH, OF=0, SF=1, ZF=0, PF=1, CF=1
neg ax          ;AX=0001H, OF=0, SF=0, ZF=0, PF=0, CF=1
dec al          ;AX=0000H, OF=0, SF=0, ZF=1, PF=1, CF=1
neg ax          ;AX=0000H, OF=0, SF=0, ZF=1, PF=1, CF=0

```

验证各条指令的执行结果以及对标志的影响，可以参考图 3-6 那样单步执行。虽然读者可能感觉仅仅进行重复性的调试操作似乎没有意义，但通过这些示例的实践可以让大家熟练掌握调试程序的使用，为进一步进行完整程序的调试打好基础。

5. 比较指令 CMP

比较指令 CMP (Compare) 将目的操作数减去源操作数，但差值不回送目的操作数，只按照减法结果影响状态标志，格式如下：

```

cmp reg,imm/reg/mem    ;减法: reg-imm/reg/mem
cmp mem,imm/reg         ;减法: mem-imm/reg

```

CMP 指令通过减法运算影响状态标志，根据标志状态可以获知两个操作数的大小关系。它主要给条件转移等指令使用其形成的状态标志（下章学习）。

3.2.4 乘法和除法指令

8086 处理器的乘法和除法指令需要区别无符号数和有符号数，并隐含使用了 AX(和 DX) 寄存器，学习时需要予以注意。

1. 乘法指令 MUL/IMUL

乘法指令指出源操作数 `reg/mem`（寄存器或存储单元），隐含使用目的操作数 `AX`（和 `DX`），如表 3-4 所示。如果给定源操作数是 8 位数 `r8/m8`，`AL` 与其相乘得到 16 位积，存入 `AX` 中；若是 16 位数 `r16/m16`，`AX` 与其相乘，得到 32 位积，高 16 位存入 `DX`，低 16 位存入 `AX` 中。`DX` 和 `AX` 作为一个寄存器对表达 32 位数据，常书写为 `DX:AX`，中间有一个小数点。

表 3-4 乘法指令

指令类型	指 令	操作数组合及功能	举 例
无符号数乘法	<code>mul reg/mem</code>	8 位乘法： $AX = AL \times r8/m8$ 16 位乘法： $DX:AX = AX \times r16/m16$	<code>mul bl</code>
有符号数乘法	<code>imul reg/mem</code>		<code>imul bx</code> <code>mul wvar</code>

乘法指令分成无符号数乘法指令 `MUL` 和有符号数乘法指令 `IMUL`。因为同一个二进制编码表示无符号数和有符号数时，真值可能不同。

例如，用 `MUL` 进行 8 位无符号乘法运算：

```
mov al,0a5h      ;AL=A5H，作为无符号整数编码，表示真值：165
mov bl,64h       ;BL=64H，作为无符号整数编码，表示真值：100
mul bl           ;无符号乘法：AX=4074H，表示真值：16500
```

如果，用 `IMUL` 进行 8 位有符号乘法运算：

```
mov al,0a5h      ;AL=A5H，作为有符号整数补码，表示真值：-91
mov bl,64h       ;BL=64H，作为有符号整数补码，表示真值：100
imul bl          ;有符号乘法：AX=DC74H，表示真值：-9100
```

所以，计算二进制数乘法 $A5H \times 64H$ 时，如果把它们当做无符号数，用 `MUL` 指令，其结果为 `4074H`，表示真值 16500；如果采用 `IMUL` 指令，则结果为 `DC74H`，表示真值 -9100。

注意：加减指令只进行无符号数运算，程序员利用 `CF` 和 `OF` 区别结果。

乘法指令按如下规则影响标志 `OF` 和 `CF`。若乘积的高一半是低一半的符号位扩展，说明高一半不含有效数值，则 $OF=CF=0$ ；乘积高一半有效，则用 $OF=CF=1$ 表示。设置 `OF` 和 `CF` 标志的原因，是我们有时需要知道高一半是否可以被忽略，而不影响结果（下文解释）。

但是，乘法指令对其他状态标志没有定义，即为任意，不可预测。注意，这一点是与数据传送类指令对标志没有影响不同的，没有影响是指不改变原来的状态。

2. 除法指令 DIV/IDIV

除法指令给出源操作数 `reg/mem`（寄存器或存储单元），隐含使用目的操作数 `AX`（和 `DX`），如表 3-5 所示。

表 3-5 除法指令

指令类型	指 令	操作数组合及功能	举 例
无符号数除法	<code>div reg/mem</code>	8 位除法： $AL = AX \div r8/m8$ 的商， $AH = AX \div r8/m8$ 的余数	<code>div cl</code> <code>div cx</code>
有符号数除法	<code>idiv reg/mem</code>	16 位除法： $AX = DX:AX \div r16/m16$ 的商， $DX = DX:AX \div r16/m16$ 的余数	<code>idiv bvar</code> <code>idiv wvar[si]</code>

类似乘法指令，除法指令也隐含使用 `AX`（和 `DX`），并且被除数的位数要倍长于除数的

位数。除法指令也分成无符号除法指令 DIV 和有符号除法指令 IDIV。有符号除法时，余数的符号与被除数的符号相同。对同一个二进制编码，分别采用 DIV 和 IDIV 指令后，除商和余数也会不同。

例如，用 DIV 进行 8 位无符号除法运算：

```
mov ax,400h      ;AX=400H，作为无符号整数编码，表示真值：1024
mov bl,0b4h      ;BL=B4H，作为无符号整数编码，表示真值：180
div bl           ;无符号除法：商 AL=05H，余数：AH=7CH（真值：124）
                ;表示计算结果：5×180+124=1024
```

如果用 IDIV 进行 8 位有符号除法运算：

```
mov ax,400h      ;AX=400H，作为有符号整数补码，表示真值：1024
mov bl,0b4h      ;BL=B4H，作为有符号整数补码，表示真值：-76
idiv bl          ;有符号除法：商 AL=F3H（真值：-13），余数：AH=24H（真值：36）
                ;表示计算结果：(-13)×(-76)+36=1024
```

除法指令使状态标志没有定义，但是却可能产生除法溢出。除数为 0，或者商超过了所能表达的范围，则发生除法溢出。用 DIV 指令进行无符号数除法，商所能表达的范围是：字节量除时为 0~255，字量除时为 0~65535，双字量除时为 $0 \sim 2^{32}-1$ 。用 IDIV 指令进行有符号数除法，商所能表达的范围是：字节量除时为 -128~127，字量除时为 -32768~32767，双字量除时为 $-2^{31} \sim 2^{31}-1$ 。发生除法错溢出，8086 处理器将产生编号为 0 的内部中断。实用的程序中应该考虑这个问题，操作系统通常只会提示错误。

图 3-7 演示除法指令的调试过程，操作步骤简述如下：

```
D:\ML615>debug
0059:0100 div bl
0059:0102
-rax
AX 0000
:400
-rbx
BX 0000
:t
t
AX=7C05 BX=00B4 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0102 NU UP EI PL NZ NA PO NC
0059:0102 50          PUSH    AX
-rax
0059:0102 idiv bl
0059:0104
-rax
AX 7C05
:400
-t
AX=24F3 BX=00B4 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D59 ES=0D59 SS=0D59 CS=0D59 IP=0104 NU UP EI PL NZ NA PO NC
0059:0104 800E569180 OR      BYTE PTR [9156],80 DS:9156=00
-rbx
BX 00B4
:t
t=102
Divide overflow
D:\ML615>
```

图 3-7 除法指令的调试截图

(1) 验证无符号除法指令，在 DEBUG 调试程序中汇编“DIV BL”指令。

(2) 利用寄存器命令 R 直接给寄存器 AX 和 BX 赋值。在调试程序提示符下，输入“RAX”命令将显示寄存器 AX 当前的状态，在下一行则可以输入新值（本例是 400H）。同样方法给 BX 赋值（本例是 B4H）。R 命令可以对 16 位寄存器赋值，不能直接对 8 位通用寄存器赋值。

(3) 执行刚才汇编的无符号除法指令，观察 AX 值，其中低字节 AL 保存商，高字节 AH 保存余数。

(4) 接着，使用同样的方法验证有符号除法指令“IDIV BL”。进入 DEBUG 后，使用不带地址参数的汇编命令 A 默认从偏移地址 0100H 开始保存汇编的指令代码。再次使用 A 命令，如果仍然不带参数，则默认接着上次的地址保存汇编的指令。所以，本例的“IDIV BL”

被安排在 0102H 地址位置。

(5) 不带参数使用跟踪命令 T 按照当前 CS: IP 值执行, 执行一条指令后 IP 值自动增加该指令的字节数, 再次执行 T 命令自然接着上次执行的指令。如果不是按照当前 IP 指定的地址执行指令, 需要使用地址参数说明。

(6) 本例后面设置 BX 为 0, 使用“T 102”命令再次执行在 0102H 地址位置的“IDIV BL”指令。不过, 由于除数 BL 为 0, 所以出现了除法错。DOS 发现了这个错误, 提示除法溢出 (Divide Overflow) 信息, 并关闭 DEBUG 程序, 防止其继续错误下去。

【例 3-6】 温度转换程序。

摄氏温度 C 转换为华氏温度 F 的公式是: $F=(9/5) \times C+32$, 温度值用 16 位整数表示。

```

;数据段
tempc    dw 26          ;假设一个摄氏温度 C
tempf    dw ?           ;保存华氏温度 F
;代码段
mov ax,tempc    ;取摄氏温度值: AX=C
mov bx,9
imul bx        ;DX.AX=C×9, 作为被除数
mov bx,5
idiv bx        ;AX=C×9/5 (没有考虑余数)
add ax,32      ;AX=F=C×9/5+32
mov tempf,ax   ;保存华氏温度值 F
```

本例程序没有显示结果, 读者可以利用子程序库实现。还建议读者利用调试程序动态跟踪指令执行, 观察指令的执行情况 (参见附录 A)。

3.2.5 其他运算指令

除基本的二进制加减乘除指令外, 算术运算指令还包括十进制 BCD 码运算以及与运算相关的符号扩展等指令。

1. 符号扩展指令

8086 处理器支持 8 位和 16 位数据操作, 大多数指令要求两个操作数类型一致。但是, 实际的数据类型不一定满足要求。例如, 16 位与 8 位数据的加减运算, 需要先将 8 位扩展为 16 位。再如, 16 位除法需要将被除数扩展成 32 位。不过, 位数扩展后数据大小不能因此改变。

对无符号数据, 只要在前面加 0 就可实现位数扩展、大小不变, 这就是零位扩展 (Zero Extension)。例如, 8 位无符号数据 80H (=128), 零位扩展为 16 位 0080H (=128)。

8086 处理器没有设计实现零位扩展的指令, 需要时可以直接对高位进行赋值 0 实现。例如, 在例 3-3 不使用 XLAT 指令的程序中:

```

mov al,[si]      ;AL←要转换的数字
mov ah,0         ;AH←0
add bx,ax        ;BX←BX+AX, 指向对应的字符
```

为了实现 16 位偏移地址相加, 赋值 AH 等于 0, 使得在 AL 的 8 位数据扩展为 16 位存入 AX, 这样就可以与 16 位寄存器 BX 内容相加。也许你会考虑使用指令“ADD BX, AL”, 但这是一条非法指令。或者你可能使用 8 位加法指令“ADD BL, AL”, 这是一条正确的指令, 但如果进位, 进位并不会自动进入 BH 当中, 结果还会出错。

对有符号数据（补码）表示，增加位数而保持数据大小不变，需要进行符号扩展（Flag Extension），即用一个操作数的符号位（即最高位）形成另一个操作数，增加的各位全部是符号位的状态。例如，8 位有符号数据 64H（=100）为正数，符号位为 0，（高位）符号扩展成 16 位是 0064H（=100）。再如，16 位有符号数据 FF00H（=-256）为负数，符号位为 1，符号扩展成 32 位是 FFFFFFF00H（=-256）。还有一个典型的例子是真值“-1”，字节量补码表达是 FFH，字量补码是 FFFFH，双字量补码表达为 FFFFFFFFH。

8086 处理器设计有两条符号扩展指令 CBW 和 CWD：

```
cbw          ;AL 符号扩展到 AX
cwd          ;AX 符号扩展到 DX 和 AX 寄存器对 (DX:AX)
```

例如：

```
mov al,82h   ;AL=82H
cbw          ;符号扩展：AX=FF82H
add al,0fh   ;AL=72H=82H+F0H（进位在 CF 标志中）
cbw          ;符号扩展：AX=0072H
cwd          ;符号扩展：DX:AX=00000072H
```

整数数据经过零位或者符号扩展增加了位数，大小没有变化，新扩展的位数只是数据的符号，并没有数值含义。反过来说，如果高位部分都是符号位，可以截断这些高位部分，也不改变数据大小。例如真值-1，用 32 位补码表达为 FFFFFFFFH，高位都是符号位，所以截断高 16 位得到 16 位表达是 FFFFH；其实“-1”用 8 位就可以表达了，所以还可以再截断高 8 位。

这时，反过来理解乘法指令对标志 OF 和 CF 影响的设计原因。两个 N 位数据相乘，可能得到 $2N$ 位的乘积。但如果乘积的高一半是低一半的符号位扩展，说明高一半不含有有效数值，就可以放心地截断高一半而不影响正确的结果。

2. 十进制调整指令

十进制数在计算机中也要用二进制编码表示，这就是二进制编码的十进制数 BCD 码。前述算术运算指令实现了二进制数的加减乘除，要实现十进制 BCD 码运算，还需要对二进制运算结果进行调整。这是因为 4 位二进制码有 16 种编码代表 0~F，而 BCD 码只使用其中 10 种编码代表 0~9；当 BCD 码按二进制运算后，不可避免地会出现 6 种不用的编码；十进制调整指令就是在需要时让二进制结果跳过这 6 种不用的编码，而仍以 BCD 码反映正确的 BCD 码运算结果。

例如， $9+6=15$ ，8 位二进制运算是 $00001001+00000110=00001111$ ，BCD 码指令调整结果为 00010101，即表示 15 的 BCD 码。

8086 处理器支持压缩 BCD 码调整指令和非压缩 BCD 码调整指令。压缩 BCD 码就是通常的 8421 码，它用 4 个二进制位表示一个十进制位，一个字节可以表示两个十进制位，即 00~99。DAA 和 DAS 指令分别实现加法和减法的压缩 BCD 码调整。

非压缩 BCD 码用 8 个二进制位表示一个十进制位，实际上只是用低 4 个二进制位表示一个十进制位 0~9，高 4 位任意（建议总设置为 0，以免出错）。ASCII 码中 0~9 的编码是 30H~39H，所以 0~9 的 ASCII 码（高 4 位变为 0）就可以认为是非压缩 BCD 码。AAA、AAS、AAM 和 AAD 指令依次实现非压缩 BCD 码的加减、乘、除法调整。

3.3 位操作类指令

计算机中最基本的数据单位是二进制位，指令系统设计有针对二进制位进行操作、实现位控制的指令。当需要进行一位或若干位的处理时，可以考虑采用位操作类指令。

3.3.1 逻辑运算指令

正像数学中的算术运算，逻辑运算是逻辑代数的基本运算。逻辑与门电路，逻辑或门电路以及逻辑非门电路也是数字电路最基本的物理器件。

1. 逻辑与指令 AND

逻辑与运算规则是：进行逻辑与运算的两位都是逻辑 1，则结果是 1；否则，结果是 0。也就是说，逻辑 0 和逻辑 0 或逻辑 1 相与结果都为 0，只有逻辑 1 和逻辑 1 相与结果才为 1。这个规则类似二进制的乘法，所以也称其为逻辑乘。在逻辑代数中常采用乘法的运算符号“ \cdot ”表示逻辑与，一般则使用“ \wedge ”表示逻辑与。图 3-8 给出了逻辑与的真值表、逻辑表达式及运算示例。真值表是数字逻辑中经常采用的表达输入与输出关系的功能表。

真值表			逻辑表达式 $T = A \cdot B$	
输入		输出		
A	B	T		
0	0	0		
0	1	0		
1	0	0		
1	1	1		

示例

	01000101
\wedge	00110001
	00000001

图 3-8 逻辑与的真值表、表达式及运算示例

逻辑与指令 AND 将两个操作数按位进行逻辑与运算，结果返回目的操作数，格式如下：

and reg,imm/reg/mem ;逻辑与: $\text{reg} = \text{reg} \wedge \text{imm/reg/mem}$
and mem,imm/reg ;逻辑与: $\text{mem} = \text{mem} \wedge \text{imm/reg}$

AND 指令支持的目的操作数是寄存器和存储单元，源操作数是立即数、寄存器和存储单元，二者不能都是存储器操作数。它设置标志 $\text{CF} = \text{OF} = 0$ ，根据结果按定义影响 SF、ZF 和 PF。

2. 逻辑或指令 OR

逻辑或运算规则是：进行逻辑或运算的两位都是逻辑 0，则结果是 0；否则，结果是 1。也就是说：只有逻辑 0 和逻辑 0 相或结果才为 0，逻辑 0 和逻辑 1 或逻辑 1 相或结果都为 1。这个规则有点像无进位的二进制加法，所以也称其为逻辑加。在逻辑代数中常采用加法的运算符号“ $+$ ”表示逻辑或，一般则使用“ \vee ”表示逻辑或。图 3-9 给出了逻辑或的真值表、逻辑表达式及运算示例。

真值表			逻辑表达式 $T = A + B$	
输入		输出		
A	B	T		
0	0	0		
0	1	1		
1	0	1		
1	1	1		

示例

	01000101
\vee	00110001
	01110101

图 3-9 逻辑或的真值表、表达式及运算示例

逻辑或指令 OR 将两个操作数按位进行逻辑或运算，结果返回目的操作数，格式如下：

or reg,imm/reg/mem ;逻辑或: $\text{reg} = \text{reg} \vee \text{imm/reg/mem}$

or mem,imm/reg ;逻辑或: $\text{mem} = \text{mem} \vee \text{imm/reg}$

OR 指令支持目的操作数是寄存器和存储单元，源操作数是立即数、寄存器和存储单元，但二者不能都是存储器操作数。它设置标志 $\text{CF} = \text{OF} = 0$ ，根据结果按定义影响 SF、ZF 和 PF。

3. 逻辑非指令 NOT

逻辑非运算针对一个位进行求反，规则是原来为 0 的位变成 1，原来为 1 的位变成 0，所以也称逻辑反。在逻辑代数中常采用加上画线“ \neg ”表示对其进行求反，一般则使用“ \sim ”表示逻辑非。图 3-10 给出了逻辑非的真值表、逻辑表达式及运算示例。数字电路中常用一个小圆表示求反或者低电平有效。

真值表		逻辑表达式 $T = \bar{A}$
输入	输出	
A	T	示例
0	1	\sim 01000101
1	0	00110001

图 3-10 逻辑非的真值表、表达式及运算示例

逻辑非指令 NOT 是单操作数指令，按位进行逻辑非运算，结果返回，格式如下：

not reg/mem ;逻辑非: $\text{reg/mem} = \sim \text{reg/mem}$

NOT 指令支持的操作数是寄存器和存储单元，不影响标志位。

4. 逻辑异或指令 XOR

逻辑异或运算规则是，进行逻辑异或运算的两位相同，则结果是 0；否则，结果是 1。也就是说，逻辑 0 和逻辑 0 相异或结果为 0，逻辑 0 和逻辑 1 相异或结果为 1，逻辑 1 和逻辑 0 相异或结果为 1，逻辑 1 和逻辑 1 相异或结果为 0。这个规则更像不考虑进位的二进制加法，所以也称其为逻辑半加。在逻辑代数中常采用“ \oplus ”表示逻辑异或。图 3-11 给出了逻辑异或的真值表、逻辑表达式及运算示例。

真值表			逻辑表达式 $T = A \oplus B$
输入		输出	
A	B	T	示例
0	0	0	01000101
0	1	1	\oplus 00110001
1	0	1	01110100
1	1	0	

图 3-11 逻辑异或的真值表、表达式及运算示例

逻辑异或指令 XOR 将两个操作数按位进行逻辑异或运算，结果返回目的操作数。XOR 指令支持的操作数组合、对标志的影响与 AND、OR 指令一样。格式如下：

xor reg,imm/reg/mem ;逻辑异或: $\text{reg} = \text{reg} \oplus \text{imm/reg/mem}$

xor mem,imm/reg ;逻辑异或: $\text{mem} = \text{mem} \oplus \text{imm/reg}$

【例 3-7】 逻辑运算程序。

```

;数据段
varA    dw 0101010101001101b
varB    dw 0011010111100001b
varT1   dw ?

```

```

varT2    dw ?
          ;代码段
          mov ax,varA          ;AX=0101010101001101B
          not ax                ;AX=1010101010110010B
          and ax,varB          ;AX=0010000010100000B
          mov bx,varB          ;BX=0011010111100001B
          not bx                ;BX=1100101000011110B
          and bx,varA          ;BX=0100000000001100B
          or ax,bx             ;AX=0110000010101100B
          mov varT1,ax
          ;
          mov ax,varA
          xor ax,varB          ;AX=0110000010101100B
          mov varT2,ax

```

基本的逻辑运算是与、或、非，逻辑异或可以书写成如下逻辑表达式：

$$A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$$

本例程序的前一段（8 条指令）将 VARA 和 VARB 表达的逻辑变量按照上述公式的右侧进行运算，结果保存在 VART1 中。接着用异或指令实现 VARA 和 VARB 的异或运算，将结果保存在 VART2。所以本例程序运行后 VART1 和 VART2 内容相同。

逻辑运算指令除可进行逻辑运算外，经常用于设置某些位为 0 或 1 或求反。AND 指令可用于复位某些位（同“0”与），但不影响其他位（同“1”与）。OR 指令可用于置位某些位（同“1”或），而不影响其他位（同“0”或）。XOR 可用于求反某些位（同“1”异或），而不影响其他位（同“0”异或）。

例如：

```

and bl,11110110b    ;BL 中 D3 和 D0 位被清 0，其余位不变
or bl,00001001b     ;BL 中 D3 和 D0 位被置 1，其余位不变
xor bl,00001001b     ;BL 中 D3 和 D0 位被求反，其余位不变

```

XOR 指令常用于将某个寄存器清 0，同时还使 CF=OF=0。这与用减法进行自身相减得到同样的结果。而直接传送 0 进入寄存器也可以清 0，不过状态标志没有被改变。

对比如下，哪个最好：

```

xor dx,dx            ;DX=0, CF=OF=0, SF=0, ZF=1, PF=1
sub dx,dx            ;DX=0, CF=OF=0, SF=0, ZF=1, PF=1
mov dx,0             ;DX=0, 状态标志不变

```

大小写字母的 ASCII 值相差 20H，利用“SUB BL,20H”指令可以实现小写字母转换为大写字母；利用“ADD BL,20H”指令可以实现大写字母转换为小写字母。通过 ASCII 码表，还可以观察到大写字母与小写字母仅 D5 位不同，例如大写字母“A”的 ASCII 码值为 41H（01000001B）、D5=0；而小写字母“a”=61H（01100001B）、D5=1。所以，利用逻辑运算指令也非常容易实现大小写转换。

例如（假设 DL 寄存器内是小写或大写字母）：

```

and dl,11011111b    ;小写转换为大写：D5 位清 0，其余位不变
or dl,00100000b      ;大写转换为小写：D5 位置 1，其余位不变
xor dl,00100000b     ;大小写互相转换：D5 位求反，其余位不变

```


5. 测试指令 TEST

测试指令 TEST 将两个操作数按位进行逻辑与运算，格式如下：

```
test reg,imm/reg/mem    ;逻辑与: reg ^ imm/reg/mem
test mem,imm/reg         ;逻辑与: mem ^ imm/reg
```

TEST 指令不返回逻辑与结果，只根据结果像 AND 指令一样来设置状态标志。TEST 指令通常用于检测一些条件是否满足，但又不希望改变原操作数的情况。TEST 指令和 CMP 指令类似，一般后跟条件转移指令，目的是利用测试条件转向不同的分支（详见下一章）。

3.3.2 移位指令

移位指令将数据以二进制位为单位向左或向右移动，有多种处理移入和移出位的方式。

1. 移位指令

移位（Shift）指令分逻辑（Logical）移位和算术（Arithmetic）移位，分别具有左移（Left）或右移（Right）操作功能，如图 3-12 所示，指令格式如下：

```
shl reg/mem,1/CL    ;逻辑左移: reg/mem 左移 1/CL 位，最低位补 0，最高位进入 CF
shr reg/mem,1/CL    ;逻辑右移: reg/mem 右移 1/CL 位，最高位补 0，最低位进入 CF
sal reg/mem,1/CL    ;算术左移，与 SHL 是同一条指令
sar reg/mem,1/CL    ;算术右移: reg/mem 右移 1/CL 位，最高位不变，最低位进入 CF
```

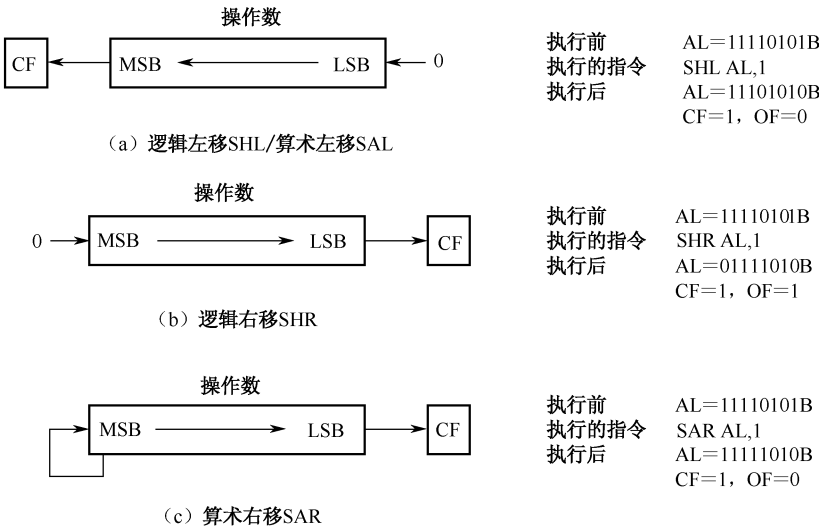


图 3-12 移位指令的功能和示例

4 条（实际为 3 条）移位指令的目的操作数可以是寄存器或存储单元。后一个操作数表示移位位数，用“1”表示移动一位，用 CL 表示移动 CL 寄存器值的次数。

移位指令根据最高或最低移出的位设置进位标志 CF，根据移位后的结果影响 SF, ZF 和 PF。如果进行一位移动，则按照操作数的最高符号位是否改变，相应设置溢出标志 OF；如果移位前的操作数最高位与移位后操作数的最高位不同（有变化），则 OF=1；否则 OF=0。当移位次数大于 1 时，OF 不确定。

逻辑移位指令可以实现无符号数的乘以或除以 2、4、8……。SHL 指令执行一次逻辑左

移位，原操作数每位的权增加了一倍，相当于乘 2；SHR 指令执行一次逻辑右移位，相当于除以 2，商在操作数中，余数由 CF 标志反映。

【例 3-8】 移位指令实现乘法程序。

```
      ;数据段
bvar  db 64
      ;代码段
xor ax,ax      ;AX=0
mov al,bvar     ;AL=要乘以 10 的无符号数
shl ax,1        ;左移一位等于乘 2
mov bx,ax       ;BX=AX×2
shl ax,1        ;再左移 2 位，AX=AX×8
shl ax,1
add ax,bx       ;AX=AX×10
;
mov bx,10
mul bx          ;DX.AX=AX×10
```

本例程序先将 BVAR 变量保存的无符号整数值扩大 10 倍（未使用乘法指令），然后再乘以 10（使用乘法指令）。第 1 段程序没有使用乘法指令，而是用逻辑左移一位等于乘 2，再左移 2 位实现乘 8，然后 2 倍数据与 8 倍数据相加获得 10 倍数据。虽然这种算法比第 2 段直接使用乘法指令烦琐，但是在简单的没有乘除法指令的处理器中非常实用。即使在有乘除法指令的处理器中，这种算法的程序执行速度仍然快于使用乘法指令。这是因为移位指令、加减指令都使用非常简单的硬件逻辑实现、执行速度很快；相比而言，实现乘除法的硬件电路比较复杂、执行速度较慢。例如在 16 位 8086 处理器中执行乘法需要一百个以上的时钟周期，加减法指令和移位指令只有几个时钟周期。

运行结果的判断，可以进入调试程序，或者利用本书提供的 DISPUIW 子程序，实现以无符号十进制形式显示 AX 内容。

2. 循环移位指令

循环（Rotate）移位指令类似移位指令，但要将从一端移出的位返回到另一端形成循环。它分成不带进位循环移位和带进位循环移位，分别具有左移和右移操作，如图 3-13 所示，指令格式如下：

```
rol reg/mem,1/CL
;不带进位循环左移：reg/mem 左移 1/CL 位，最高位进入 CF 和最低位
ror reg/mem,1/CL
;不带进位循环右移：reg/mem 右移 1/CL 位，最低位进入 CF 和最高位
rcl reg/mem,1/CL
;带进位循环左移：reg/mem 左移 1/CL 位，最高位进入 CF，CF 进入最低位
rcr reg/mem,1/CL
;带进位循环右移：reg/mem 右移 1/CL 位，最低位进入 CF，CF 进入最高位
```

循环移位指令的操作数形式与移位指令相同，按指令功能设置进位标志 CF，但不影响 SF、ZF、PF 标志。对 OF 标志的影响，循环移位指令与前面介绍的移位指令一样。

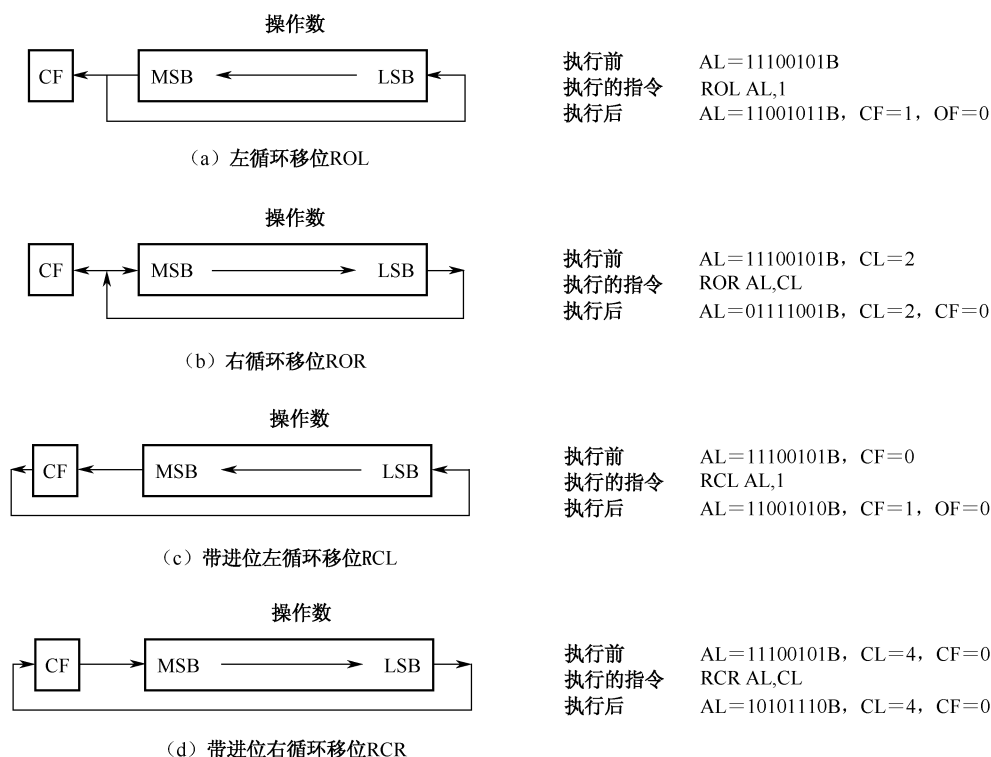


图 3-13 循环移位指令的功能和示例

【例 3-9】 循环移位程序。

```

;数据段
dvar    dd 12346789h
ascii   db '38'
bcd      db ?

;代码段
mov cx,4
again:  shr word ptr dvar+2,1  ;先移动高 16 位
        rcr word ptr dvar,1    ;后移动低 16 位
        loop again
;
mov al,ascii
and al,0fh                ;处理低 4 位对应的字符
mov ah,ascii+1
mov cl,4
shl ah,cl                  ;处理高 4 位对应的字符
or al,ah                  ;组合形成压缩 BCD 码
mov bcd,al

```

8086 处理器可以直接对 8 位和 16 位数据进行各种移位操作，但是对多于 16 位的数据就需要组合移位指令实现。本例程序将 DVAR 指定的 32 位数据逻辑右移 4 位。首先可以将高 16 位逻辑右移一位（用 SHR 指令），最高位被移入 0，移出的位进入了标志 CF；接着用带进位右移一位（用 RCR 指令），这样 CF 内容（即高 16 位移出的位）进入到低 16 位，同时最

低位进入 CF，这样就实现了 32 位数据右移一位，如图 3-14 所示。需要多少位移动，就设置 CX 等于多少，用循环指令 LOOP 实现多少次循环就可以了。

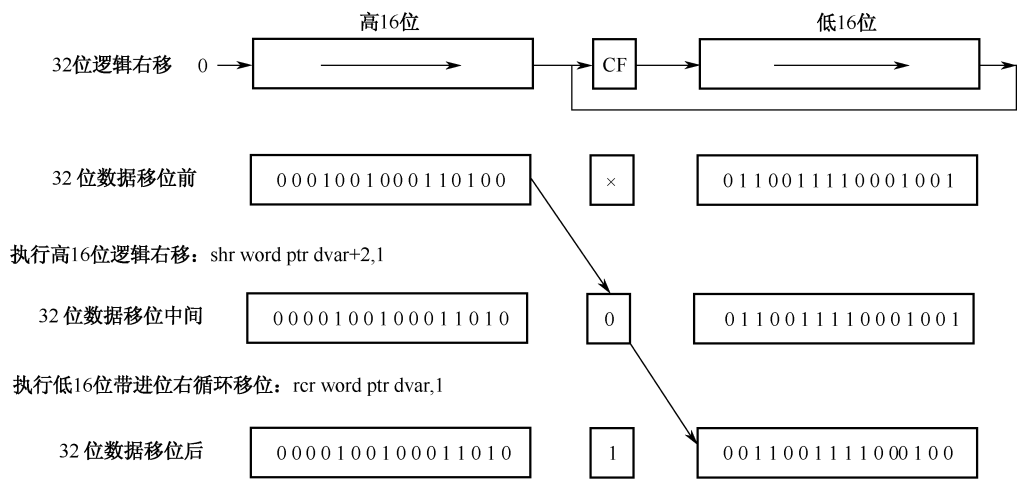


图 3-14 32 位数据的移位

底层程序设计中，经常要将数据在不同编码间进行相互转换，这时利用位操作类指令很方便。本例后部分程序将两个 ASCII 码转换为压缩 BCD 码。先将低字节 ASCII 码取出，只保留表示数值的低 4 位、高 4 位清 0；然后取出高字节 ASCII 码左移 4 位，使表示数值的 4 位移到高 4 位、同时低 4 位被移入 0；最后，用逻辑或指令合并高 4 位和低 4 位数值，转换成为 BCD 码。这样，从低地址到高地址依次存放的两个字符“3”和“8”（类似从键盘依次输入），按照低地址对应低字节、高地址对应高字节原则，转换为 BCD 码 83H。

习 题 3

3.1 简答题

- (1) 如何修改“MOV SI, BYTE PTR 250”语句使其正确？
- (2) 为什么说“XCHG DX, CL”是一条错误的指令？
- (3) 解释 8086 处理器的堆栈段“向下生长”是什么意思？
- (4) 都是获取偏移地址，为什么指令“LEA BX, [SI]”正确，而指令“MOV BX, OFFSET[SI]”错误？
- (5) 执行了一条加法指令后，发现 ZF=1，说明结果是什么？
- (6) INC, DEC, NEG 和 NOT 都是单操作数指令，这个操作数应该是源操作数还是目的的操作数？
- (7) 大小写字母转换使用了什么规律？
- (8) 乘除法运算针对无符号数和有符号数，有两种不同的指令。只有一种指令的加减法运算如何区别无符号数和有符号数运算？
- (9) 除法指令“DIV SI”的被除数是什么？
- (10) 逻辑与运算为什么也称为逻辑乘？

3.2 判断题

- (1) 指令“MOV AX,0”使 AX 结果为 0，所以标志 ZF=1。
- (2) 空操作 NOP 指令其实根本没有指令。
- (3) 堆栈的操作原则是“先进后出”，所以堆栈段的数据除 PUSH 和 POP 指令外，不允许其他方法读写。
- (4) 虽然 ADD 指令和 SUB 指令执行后会影响到标志状态，但执行前的标志并不影响它们的执行结果。
- (5) 80 减 90 (80-90) 需要借位，所以执行结束后，进位标志 CF=1。
- (6) 指令“INC CX”和“ADD CX,1”的实现功能完全一样（包括标志），可以互相替换。
- (7) 无符号数在前面加零扩展，数值不变；有符号数前面进行符号扩展，位数加长一位、数值增加一倍。
- (8) CMP 指令是目的操作数减去源操作数，与 SUB 指令功能相同。
- (9) 逻辑运算没有进位或溢出问题，此时 CF 和 OF 没有作用，所以逻辑运算指令如 AND、OR 等将 CF 和 OF 设置为 0。
- (10) SHL 指令左移一位，就是乘 10。

3.3 填空题

- (1) 指令“PUSH DX”执行后，SP 会_____。
- (2) 指令“POP DX”的功能是将堆栈顶部的数据传送给_____，然后_____。
- (3) 例 3-3 的 TAB 定义如果是“1234567890”，则显示结果是_____。
- (4) 进行 8 位二进制数加法 BAH+6CH，8 位结果是_____，标志 PF=_____。
如果进行 16 位二进制数加法 45BAH+786CH，16 位结果是_____，标志 PF=_____。
- (5) 假设 AX=98H，执行“NEG AX”指令后，AX=_____，标志 SF=_____。
- (6) 假设 AX=98H，执行“CBW”指令后，AX=_____。
- (7) 假设 AX=98H，执行“CWD”指令后，DX=_____。
- (8) 指令“XOR AX,AX”和“SUB AX,AX”执行后，AX=_____，CF=OF=_____。
而指令“MOV AX,0”执行后，AX=_____，CF 和 OF 没有变化。
- (9) 例 3-9 程序执行结束，变量 DVAR 内容是_____。
- (10) 欲将 DX 内的无符号数除以 16，可以使用指令“SHR DX,_____”，其中后一个操作数是一个 8 位寄存器，其值等于_____。

3.4 MOV 指令支持多种操作数组合，请给每种组合各举一个实例。

- (1) mov reg, imm
- (2) mov mem, imm
- (3) mov reg, reg
- (4) mov mem, reg
- (5) mov seg, reg
- (6) mov reg, mem
- (7) mov seg, mem
- (8) mov reg, seg
- (9) mov mem, seg

3.5 操作数的组合通常符合逻辑，但不是任意的，指出如下指令的错误原因。

- (1) mov cx,dl
- (2) mov ip,ax
- (3) mov es,1234h
- (4) mov es,ds
- (5) mov al,300
- (6) mov [si],45h
- (7) mov ax,bx+di
- (8) mov 20h,ah

3.6 使用 MOV 指令实现交换指令“XCHG BX,[DI]”功能。

3.7 什么是堆栈，它的工作原理是什么，它的基本操作有哪两个，对应哪两种指令？

3.8 假设当前 SP=4300H，说明下面每条指令后，SP 等于多少？

```
push ax
push dx
pop ax
pop word ptr [bx]
pop bx
```

3.9 已知数字 0~9 对应的格雷码依次为：18H、34H、05H、06H、09H、0AH、0CH、11H、12H、14H；请为如下程序的每条指令加上注释，说明每条指令的功能和执行结果。

```
      ;数据段
table db 18h,34h,05h,06h,09h,0ah,0ch,11h,12h,14h
      ;代码段
      mov bx,offset table
      mov al,8
      xlat
```

为了验证你的判断，不妨使用本书的 I/O 子程序库提供的子程序 DISPHB 显示换码后 AL 的值。如果不使用 XLAT 指令，应如何修改？

3.10 举例说明 CF 和 OF 标志的差异。

3.11 分别执行如下程序片段，说明每条指令的执行结果：

(1)

```
mov ax,80h      ;AX=_____
add ax,3        ;AX=_____, CF=_____, SF=_____
add ax,80h      ;AX=_____, CF=_____, OF=_____
adc ax,3        ;AX=_____, CF=_____, ZF=_____
```

(2)

```
mov ax,100      ;AX=_____
add ax,200      ;AX=_____, CF=_____
```

(3)

```
mov ax,100      ;AX=_____
add al,200      ;AX=_____, CF=_____
```

(4)

```
mov al,7fh      ;AL=_____
sub al,8        ;AL=_____, CF=_____, SF=_____
sub al,80h      ;AL=_____, CF=_____, OF=_____
sbb al,3        ;AL=_____, CF=_____, ZF=_____
```

3.12 给出下列各条指令执行后 AL 值, 以及 CF、ZF、SF、OF 和 PF 的状态:

```
mov al,89h
add al,al
add al,9dh
cmp al,0bch
sub al,al
dec al
inc al
```

3.13 如下两段程序分别执行后, DX:AX 寄存器对的值各是多少?

(1) 加法程序

```
mov dx,11h
mov ax,0b00h
add ax,0400h
adc dx,0
```

(2) 减法程序

```
mov dx,100h
mov ax,6400h
sub ax,8400h
sbb dx,0
```

3.14 请分别用一条汇编语言指令完成如下功能:

(1) 把 BX 寄存器和 DX 寄存器的内容相加, 结果存入 DX 寄存器。

(2) 用寄存器 BX 和 SI 的基址变址寻址方式把存储器的一个字节与 AL 寄存器的内容相加, 并把结果送到 AL 中。

(3) 用 BX 和位移量 0B2H 的寄存器相对寻址方式把存储器中的一个字和 CX 寄存器的内容相加, 并把结果送回存储器中。

(4) 将 16 位变量 VARW 与数 3412H 相加, 并把结果送回该存储单元中。

(5) 把数 0A0H 与 AX 寄存器的内容相加, 并把结果送回 AX 中。

3.15 有两个 32 位无符号整数存放在变量 buffer1 和 buffer2 中, 定义数据、编写代码完成 $DX:AX \leftarrow buffer1 - buffer2$ 功能。

3.16 分别执行如下程序片段, 说明每条指令的执行结果:

(1)

```
mov si,10011100b    ;SI=_____H
and si,80h           ;SI=_____H
or si,7fh            ;SI=_____H
xor si,0feh          ;SI=_____H
```

(2)

```
mov ax,1010b         ;AX=_____B
mov cl,2
shr ax,cl             ;AX=_____B, CF=_____
shl ax,1             ;AX=_____B, CF=_____
and ax,3             ;AX=_____B, CF=_____
```

(3)

```
mov ax,1011b         ;AX=_____B
mov cl,2
```

```

rol ax,cl           ;AX=_____B, CF=_____
rcr ax,1           ;AX=_____B, CF=_____
or ax,3            ;AX=_____B, CF=_____

```

(4)

```

xor ax,ax          ;AX=_____, CF=_____, OF=_____
                  ;ZF=_____, SF=_____, PF=_____

```

3.17 给出下列各条指令执行后 AX 的结果，以及状态标志 CF、OF、SF、ZF、PF 的状态。

```

mov ax,1470h
and ax,ax
or ax,ax
xor ax,ax
not ax
test ax,0f0f0h

```

3.18 举例说明逻辑运算指令怎么实现复位、置位和求反功能？

3.19 编程将一个压缩 BCD 码变量（例如 92H）转换为对应的 ASCII 码，然后调用 DOS 的 2 号功能显示。

3.20 有 4 个 16 位有符号整数，分别保存在 VAR1、VAR2、VAR3 和 VAR4 变量中，阅读如下程序片段，得出运算公式，并说明运算结果存于何处。

```

mov ax,var1
imul var2
mov si,var3
mov di,si          ;提示：3 条指令实现 SI 数据的符号扩展到 DI
mov cl,15
sar di,cl
add ax,si
adc dx,di
sub ax,540
sbb dx,0
idiv var4

```

3.21 如下程序片段实现 AX 乘以某个数 X 的功能，请判断 X=？

请使用一条乘法指令实现上述功能。

```

mov si,ax
shl si,1
shl si,1
shl si,1
sub si,ax
mov ax,si

```

3.22 请使用移位和加减法指令编写一个程序片段计算： $AX \times 21$ ，假设乘积不超过 16 位。
提示： $21 = 2^4 + 2^2 + 2^0$ 。

3.23 阅读如下程序，为每条指令添加注释，指出其功能或作用，并说明这个程序运行后显示的结果。如果将程序中寄存器间接寻址替换为寄存器相对寻址，如何修改程序？

```

          ;数据段
num       db 6,7,7,8,3,0,0,0
tab       db '67783000'
          ;代码段

```



```

mov cx,lengthof num
mov si,offset num
mov di,offset tab
again:  mov dl,[si]
        xchg dl,[di]
        mov [si],dl
        mov ah,2
        int 21h    ;显示 AL 中的字符
        inc si
        inc di
        loop again;循环处理

```

3.24 说明如下程序执行后的显示结果:

```

;数据段
msg     db 'WELLDONE','$'
;代码段
mov cx,(lengthof msg)-1
mov bx,offset msg
again:  mov al,[bx]
        add al,20h
        mov [bx],al
        inc bx
        loop again
        mov dx,offset msg
        mov ah,9
        int 21h

```

如果将其中语句“mov bx,offset msg”改为“xor bx,bx”，则利用 BX 间接寻址的两个语句如何修改成 BX 寄存器相对寻址，就可以实现同样功能？

3.25 下面程序的功能是将数组 ARRAY1 的每个元素加固定值（500），并将保存在数组 ARRAY2。在空白处填入适当的语句或语句的一部分。

```

;数据段
array1  dw 1,2,3,4,5,6,7,8,9,10
array2  dw 10 dup(?)
;代码段
mov cx,lengthof array1
mov bx,0
again:  mov ax,array1[bx]
        add ax,500
        mov _____
        add bx,_____
        loop again

```

第4章 程序结构

使用编程语言进行程序设计，首先需要正确理解问题、分析要求，然后选择合适的数据类型和数据结构，抽象或推导出合理的实现算法，最后使用程序设计语言进行具体编码。根据具体应用情况，程序可以按照书写顺序执行，也常需要根据条件选择不同的分支，或者循环进行相同的处理，所以程序具有顺序、分支和循环3种基本结构，这也是本章的逻辑主线。但是，汇编语言通常并不直接支持结构化程序设计，而是需要使用处理器控制转移类指令实现分支、循环、调用（第5章学习）等程序结构。所以，控制转移类指令属于处理器指令系统的基本指令，也是实现分支、循环、调用等程序结构的关键指令。

4.1 顺序程序结构

顺序程序结构按照指令书写的前后顺序执行每条指令，是最基本的程序片段，也是构成复杂程序的基础，如构成分支程序的分支体、循环结构的循环体等。

【例 4-1】自然数求和程序。

知道“ $1+2+3+\cdots+N$ ”等于多少吗？自然数求和可以采用循环累加的方法，也可利用等差数列的求和公式，这样能够避免重复相加，得到改进的算法。求和公式是：

$$1+2+3+\cdots+N=(1+N)\times N\div 2$$

程序中，可以在数据段定义一个变量 NUM，作为 N 值，并预留保存求和结果的双倍长变量 SUM。以下代码段实现本例的求和运算。

为了配合调试程序详细分析整个程序的执行过程，本例给出了完整的源程序。

```
;eg401.asm
.model small
.stack
.data
num    dw 3456          ;假设一个  $N$  值（小于  $2^{16}-2$ ）
sum    dd ?
.code
.startup
mov ax,num              ;AX=N
add ax,1                ;AX=N+1
mul num                 ;DX.AX=(1+N)×N
shr dx,1                ;32 位逻辑右移一位，相当于除以 2
rcr ax,1                ;DX.AX=DX.AX÷2
mov word ptr sum,ax
mov word ptr sum+2,dx    ;按小端方式保存
.exit
end
```

程序按照公式顺序使用加法、乘法和移位指令实现加 1、乘以 N 和除以 2，最后保存结果。因为没有方便地实现数值显示的功能调用、子程序或者函数，程序没有显示结果。

1. 程序的静态分析

利用列表文件可以进行程序的静态分析，了解程序结构、标识符含义等信息。建议在使用参数“/F1”生成本例程序的列表文件时带上参数“/Sa”（注意用空格分隔前一个参数），这样可以列出汇编语句。`.startup` 和 `.exit` 代表的处理器指令，并以星号“*”标志，如下所示（后面有编号的注释是为了配合图 4-1 而加入的，其余部分省略，读者可以直接查看列表文件本身，或者参考第 1 章最后一部分的介绍）。

```
.startup
*   @Startup:
*   mov    dx, DGROUP
*   mov    ds, dx          ;设置 DS 指向数据段①
*   mov    bx, ss
*   sub    bx, dx
*   shl    bx, 001h        ;由堆栈段地址和数据段地址计算出数据段的长度②
*   shl    bx, 001h
*   shl    bx, 001h
*   shl    bx, 001h
*   cli
*   mov    ss, dx          ;设置 SS 也指向数据段，即 DS=SS③
*   add    sp, bx          ;移动栈顶偏移位置④
*   sti
*   .....
*   .exit
*   mov    ah, 04ch
*   int     021h
```

其中，`@Startup` 是 MASM 预定义的标号，用于指示程序开始执行的指令；`DGROUP` 是一个预定义的组名，指向可用区域的段地址；`CLI` 和 `STI` 分别是禁止可屏蔽中断指令和允许可屏蔽中断指令，这里配合使用的目的是避免在修改 `SS` 和 `SP` 值期间被外部中断事件打扰而引起错误。

DOS 主要采用 EXE 结构的可执行程序，它可以有独立的代码段、数据段和堆栈段，还可以有多个代码段或多个数据段，执行起始位置可以任意指定。当 DOS 执行一个程序将其装入主存时，DOS 确定当时主存最低的可用地址作为该程序的装入起始位置。这个地址以上的区域称为程序段。在该程序段偏移 0 处，DOS 为该程序建立一个程序段前缀控制块 PSP（Program Segment Prefix），占 256（100H）字节，即在偏移 100H 处才装入程序本身，如图 4-1 所示。因为源程序使用了存储模型伪指令，各段在主存的顺序默认采用标准 DOS 程序顺序，即地址从低到高依次安排代码段、数据段和堆栈段。各段之间也有默认的边界定位，一个段不必紧接着另一个段（中间可能有未用空间），段起始的偏移地址不一定是 0。

EXE 程序的加载需要重新定位：

（1）`DS` 和 `ES` 指向程序段前缀，即程序段前缀 PSP 的段地址。注意，DOS 并没有按照源程序要求将 `DS` 和 `ES` 指向设置的数据段，所以程序执行过程中需要改变 `DS` 或 `ES` 指向该程序的数据段。源程序中使用“`.startup`”语句的作用之一就是设置 `DS`，参见其生成的前两条指令（注释中有个标志^①）。

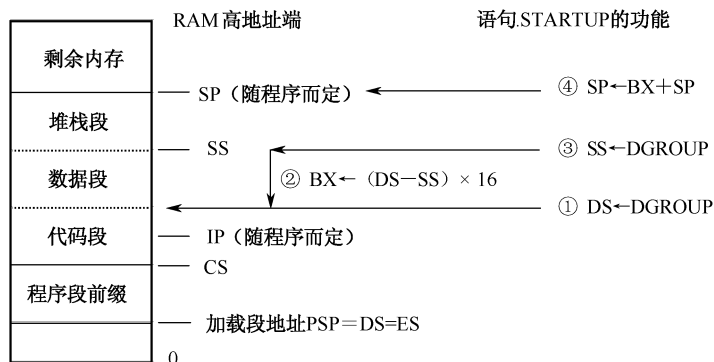


图 4-1 标准 EXE 程序的内存映象

(2) CS:IP 和 SS:SP 由连接程序确定，指向程序的代码段和堆栈段。如果不设置堆栈段，SS 指向程序段前缀，SP=100H，也就是占用程序段前缀最后的部分空间。所以，程序如果不设置堆栈段也能工作，但不能使用太大的堆栈区域。为了安全起见，程序应该设置足够的堆栈空间。

源程序使用存储模型伪指令“.model”指定程序采用小型存储模型 (SMALL)，而小型存储模型要求将程序的数据段和堆栈段组合成一个组 (Group)，共用一个段地址，即用预定义组名 DGROUP 指示的段地址。所以，“.startup”语句的另一个作用就是使堆栈段寄存器 SS 和数据段寄存器 DS 一样，都等于 DGROUP 代表的段地址。由于移动了 MS-DOS 加载该程序进入主存后设置的初始堆栈段地址，所以为了保证实际的堆栈区域没有变化，栈顶指针 SP 也需要进行相应调整，即加上数据段所占的字节数大小，参见图 4-1。由于段寄存器只保存段地址的高 16 位，以段为单位的数值需要乘以 16，也就是左移 4 位，才得到以字节为单位的主存单元数。

2. 程序的动态分析

使用调试程序可以实现程序的动态分析，一方面可以直观体会程序的动态执行过程，另一方面可以进行排查运行错误或者验证程序的正确性。

图 4-2 和图 4-3 演示自然数求和程序的调试过程，操作步骤如下。

(1) 在 DOS 环境完成源程序的编辑、汇编和连接，生成可执行文件 (EG401.EXE)。在 DOS 命令行提示符下，启动带被调试程序 (EG401.EXE) 的 DEBUG 调试程序，即输入命令“debug eg401.exe”。

```

D:\ML615>debug eg401.exe
-~
AX=0000 BX=0000 CX=0036 DX=0000 SP=0400 BP=0000 SI=0000 DI=0000
DS=0C9F ES=0C9F SS=0CB3 CS=0CAF IP=0000  NU UP EI PL NZ NA PO NC
0CAF:0000 BAB20C      MOV     DX,0CB2
-u
0CAF:0000 BAB20C      MOV     DX,0CB2
0CAF:0003 0EDA      MOV     DS,DX
0CAF:0005 8CD3      MOV     BX,SS
0CAF:0007 2BD0      SUB     BX,DX
0CAF:0009 D1E3      SHL     BX,1
0CAF:000B D1E3      SHL     BX,1
0CAF:000D D1E3      SHL     BX,1
0CAF:000F D1E3      SHL     BX,1
0CAF:0011 FA      CLI
0CAF:0012 8ED2      MOV     SS,DX
0CAF:0014 03E3      ADD     SP,BX
0CAF:0016 FB      STI
0CAF:0017 010000      MOV     AX,[0000]
0CAF:0018 03C001      ADD     AX,+01
0CAF:001D F22E0000      IMUL    WORD PTR [0000]
-d 0cb2:0 f
0CB2:0000 80 0D 00 00 00 00 00 0A 47-01 32 E4 40 D1 E0 03 D8 .....G.2.e....

```

图 4-2 自然数求和程序的调试截图 1

```

-g 17
AX=0000 BX=0010 CX=0036 DX=0CB2 SP=0410 BP=0000 SI=0000 DI=0000
DS=0CB2 ES=0C9F SS=0CB2 CS=0CAF IP=0017  NU UP EI PL NZ NA PO NC
0CAF:0017  A10000      MOV     AX,[0000]      DS:0000-0D80
-g 17
0CAF:0017  A10000      MOV     AX,[0000]
0CAF:001A  83C001      ADD     AX,+01
0CAF:001D  F72E0000     IMUL    WORD PTR [0000]
0CAF:0021  D1E8         SHR     DX,1
0CAF:0023  D1D8         RCR     AX,1
0CAF:0025  A30200      MOV     [0002],AX
0CAF:0028  89160400     MOV     [0004],DX
0CAF:002C  B44C         MOV     AH,4C
0CAF:002E  CD21         INT     21
0CAF:0030  000D00      OR      BYTE PTR [DI],00
0CAF:0033  0000         ADD     [BX+SI],AL
0CAF:0035  008A4701     ADD     [BP+SI+0147],CL
-g 2c
AX=26C0 BX=0010 CX=0036 DX=005B SP=0410 BP=0000 SI=0000 DI=0000
DS=0CB2 ES=0C9F SS=0CB2 CS=0CAF IP=002C  NU UP EI PL NZ NA PO NC
0CAF:002C  B44C         MOV     AH,4C
-d ds:0 f
0CB2:0000  00 0D C0 26 5B 00 8A 47-01 32 E4 40 D1 E0 03 D0 ...&[...G.2.E...

```

图 4-3 自然数求和程序的调试截图 2

(2) 观察被调试程序进入存储器的初始状态。在调试程序提示符下输入寄存器命令 R (可以对比不带被调试程序的初始状态截图, 参见第 2 章图 2-8)。

其中, BX 和 CX 寄存器对反映被调试程序的大小 (字节数), BX 保存高 16 位, CX 保存低 16 位。本例中, BX.CX=00000036H, 表示程序占用 54 (36H) 个存储单元。

CS:IP (本例是 0CAF:0000) 指向代码段中程序开始执行的第一条指令。

SS:SP (本例是 0CB3:0400) 指向堆栈段。因为源程序使用伪指令 “.stack” 默认分配堆栈大小是 1KB, 即从堆栈段偏移地址 0 到堆栈指针 SP 设置的区域, 所以 SP=0400H (1KB)。

DS 和 ES (本例是 0C9F) 还没有指向该程序设置的数据段, 而是指向该程序段前缀 PSP, 即该程序前 100H 位置 (段地址 0C9FH+10H=0CAFH, 注意段地址只表达 20 位物理地址的高 16 位, 所以偏移 100H 位置需要右移二进制 4 位即十六进制 1 位来参与段地址计算)。

(3) 接着查看程序的代码段指令和数据段变量初值。反汇编命令 U 可以将指定地址的内容按照指令代码解析, 并以汇编语言的格式显示其处理器指令。U 命令默认反汇编 32 字节左右的代码, 所以还可以继续使用 U 命令进行反汇编, 以便查看其余指令, 当然也可以指定地址范围的代码进行反汇编。不指定地址时, 第一次使用 U 命令默认从当前 CS:IP 开始反汇编; 接着使用不带参数的 U 命令, 则接着上次 U 命令反汇编的最后一个单元开始。

通过反汇编的第一条指令, 可以看到数据段 DS 的段地址是 0CB2H (对应列表文件的符号 DGROUP)。通过源程序获取变量值 NUM 的指令, 可以知道变量 NUM 的偏移地址 0000H。所以可以使用 D 命令查看, 但需要指明逻辑地址, 还可以给出范围 (因为只有两个变量), 即输入 “D 0CB2: 0 F”。由显示的前 2 个存储单元值得到 NUM 的初值是 0D80H (3456); 后面对应变量 SUM, 有 4 字节的预留空间, 被设置为 0。

代码段第一条指令的逻辑地址 (0CAF:0000), 转换为对应的物理地址是 0CAF0H; 变量定义后的逻辑地址 (0CB2:0006), 转换为物理地址是 0CB26H。计算占用的存储空间, 即 0CB26H-0CAF0H=36H, 就是该程序的大小 (也就是前述 BX 和 CX 寄存器对的内容)。

(4) 对程序调入主存的情况了解之后, 便可以开始进行执行, 图 4-3 继续演示程序的执行过程。使用单步执行可以观察每个指令的执行情况, 使用断点执行便于说明程序片段的执行结果。

首先使用运行命令 G 执行完成 “.startup” 语句对应的程序片段, 即输入 “G 17” 执行到断点 0017H 处暂停。观察这个程序片段的执行结果, 并对照前面的分析: DS 已经指向数据段 (段地址是 0CB2H), SS 等于 DS 指向同一个段地址, SP 也相应进行了修改, 保证堆栈区域没有改变。

(5) 接着可以单步执行，也可以断点执行到公式计算结束、退出 DOS 前的位置。

为了获得退出 DOS 前指令的地址，可以使用反汇编命令。不指明反汇编的地址，则会接着上次反汇编结束的位置继续反汇编。若要给出反汇编地址，则必须是一条指令代码的开始，否则反汇编得不到正确的指令结果，因为不论主存是什么内容，反汇编命令都按照指令代码解析其含义，显示对应的处理器指令。所以，从源程序第一条指令开始反汇编（偏移地址 0017H），即输入“U 17”命令。从反汇编的指令“mov ah, 4c”看到退出 DOS 前的地址是 002CH。

使用“G 2C”命令继续执行到公式计算结束，求和值在 DX 和 AX 寄存器对中，显示是 005B 26C0H。这个求和结果还保存在 SUM 变量中，所以还可以用 D 命令查看。由于数据段寄存器 DS 已经指向程序的数据段，故可以用“DS:”表示当前数据段的数据。按照小端方式，给出预留空间的双字长数据是 00 5B 26 C0，当然应该与 DX 和 AX 寄存器对中内容一样，表示结果：

$$1+2+3+\cdots+3456=(1+3456)\times 3456\div 2=5973696=5B26C0H$$

(6) 如果继续执行到程序退出 DOS，则 DEBUG 调试程序会显示程序终止（Program Terminated Normally）的提示信息。提醒，在调试程序中单步执行 DOS 功能调用指令“INT 21”不要用跟踪命令 T，因为命令 T 会进入到 DOS 功能调用的中断服务程序中（可以看到 CS 值改变了），最后常导致无法退出。此时，只能改用继续命令 P（Proceed），P 命令也是实现单步执行的命令，但不会进入子程序或者中断服务程序中。

高级语言的开发环境中通常也都配套调试程序，同样支持单步调试和断点调试。只是程序员多基于高级语言语句进行调试，较少进行汇编语言级调试。当然，高级语言的调试程序支持源程序级调试，可以理解变量等标识符，在图形界面下可以支持多窗口操作，使用更灵活、更方便，功能也更强大，也更复杂。DEBUG 虽然简单，但提供了最基本的调试手段，可以作为学习更复杂和更强大调试程序的基础。本书在第 8 章混合编程中将介绍 Visual C++ 6.0 配套的调试程序。

【例 4-2】 读取 CMOS RAM 数据程序。

PC 机的配置信息以及实时时钟被保存在 CMOS RAM 芯片中，系统断电后由后备电池供电，以保证信息不会丢失。CMOS RAM 有 64 字节容量，以 8 位 I/O 接口形式与处理器连接，通过两个 I/O 地址访问。访问 CMOS RAM 的内容，首先需要向 I/O 地址 70H 输出要访问的存储单元编号，然后用 I/O 地址 71H 读写该单元的 1 字节数据。

CMOS RAM 的 9、8 和 7 号字节单元依次存放着年月日数据（参见表 4-1），本例程序将它们读出并显示。这些数据的编码采用压缩 BCD 码，1 字节有 2 位 BCD 码，每个 BCD 码需要加 30H 转换为 ASCII 码，然后进行显示。

表 4-1 CMOS RAM 实时时钟信息

单元编号	含义及数值
0	秒，00H~59H 依次表示 0~59 秒
2	分，00H~59H 依次表示 0~59 分
4	时，00H~23H 依次表示 0~23 小时
6	星期，01~07H 依次表示周日、周一~周六
7	日，01H~31H 依次表示 1~31 日
8	月，01H~12H 依次表示 1~12 月
9	年，00H~99H 依次表示年份的后两位 XX00~XX99 年

从本例开始，示例程序将只给出常量说明、变量定义以及主程序或子程序代码等部分，读者需要参照第 1 章的源程序框架形成完整的源程序文件。

```

;数据段
dates    db '20xx-yy-zz','$'      ;设置日期格式，xx-yy-zz 表示年-月-日格式
;代码段
mov bx,offset dates+2             ;BX 指向日期字符串
mov cl,4                          ;设置 CL=4，用于移位指令

mov al,9                          ;AL=9（准备从 9 号单元获取年代数据）
out 70h,al                        ;从 70H 的 I/O 地址输出，选择 CMOS RAM 的 9 号单元
in al,71h                        ;从 71H 的 I/O 地址输入，获取 9 号单元的内容，保存在 AL（年代）
mov ah,al                         ;转换 AL 内容（2 位 BCD 码）为 2 个 ASCII 码
shr ah,cl                         ;转换高位
or ah,30h
mov [bx],ah                       ;保存到日期字符串中
inc bx
and al,0fh                       ;转换低位
or al,30h
mov [bx],al                       ;保存到日期字符串中
add bx,2

mov al,8                          ;AL=8（从 8 号单元获取月份数据）
out 70h,al
in al,71h
mov ah,al                         ;转换 AL 内容（2 位 BCD 码）为 2 个 ASCII 码
shr ah,cl                         ;转换高位
or ah,30h
mov [bx],ah                       ;保存到日期字符串中
inc bx
and al,0fh                       ;转换低位
or al,30h
mov [bx],al                       ;保存到日期字符串中
add bx,2

mov al,7                          ;AL=7（从 7 号单元获取日期数据）
out 70h,al
in al,71h
mov ah,al                         ;转换 AL 内容（2 位 BCD 码）为 2 个 ASCII 码
shr ah,cl                         ;转换高位
or ah,30h
mov [bx],ah                       ;保存到日期字符串中
inc bx
and al,0fh                       ;转换低位
or al,30h
mov [bx],al                       ;保存到日期字符串中
mov dx,offset dates              ;显示

```

```
mov ah,9  
int 21h
```

CMOS RAM 保存着系统的配置信息，除了上述实时时钟单元外，不要向其他单元写入内容，以免引起系统错误。

4.2 分支程序结构

基本程序块是只有一个入口和一个出口、不含分支的顺序执行程序片段。实际上，在机器语言或汇编语言中，这样的基本程序块通常由 3~5 条指令组成。改变程序执行顺序、形成分支、循环、调用等程序结构是很常见的程序设计问题。

高级语言采用 IF 等语句表达条件，并根据条件是否成立转向不同的程序分支。汇编语言需要首先利用比较 CMP、测试 TEST、加减运算、逻辑运算等影响状态标志的指令形成条件，然后利用条件转移指令判断由标志表达的条件，并根据标志状态控制程序转移到不同的程序段。

4.2.1 无条件转移指令

程序代码由机器指令组成，被安排在代码段中。代码段寄存器 CS 指出代码段的段基址，指令指针寄存器 IP 给出将要执行指令的偏移地址。随着程序代码的执行，指令指针 IP 的内容会相应改变。当程序顺序执行时，处理器根据被执行指令的字节长度自动增加 IP。但是，当程序从一个位置换到另一个位置执行指令时，IP 会随之改变，如果换到了另外一个代码段中，CS 也将相应改变。换句话说，当改变 IP（或者还包括 CS）就改变了程序的执行顺序，即实现了程序的控制转移。

1. 转移范围

程序转移的范围（远近）在 8086 处理器中有段内和段间两种。

(1) 段内转移

段内转移是指在当前代码段范围内的程序转移，因此不需要更改代码段寄存器 CS 内容，只要改变指令指针寄存器 IP 的偏移地址。段内转移相对较近，故也被称为近转移（Near）。

多数的程序转移都是在同一个代码段中，大多数的转移范围实际上很短，往往在当前位置前后不足 100 字节。如果转移范围可以用 1 字节编码表达，即向地址增大方向转移 127 字节，向地址减小方向转移 128 字节之间的距离，则形成所谓的短转移（Short）。短转移的引入是为了减少转移指令的代码长度，进而减少程序代码量。

(2) 段间转移

段间转移是指程序从当前代码段跳转到另一个代码段，此时需要更改代码段寄存器 CS 内容和指令指针 IP 的偏移地址。段间转移可以在整个存储空间内跳转、相对较远，故也被称为远转移（Far）。

2. 指令寻址方式

程序转移是处理器从当前指令跳转到目的地指令执行的过程，目的地指令所在的存储器地址被称为目的地址、目标地址或转移地址。指令寻址就是获取转移目的地址，进而执行目的地指令的方式，也称为目标地址寻址。8086 处理器设计有相对、直接和间接 3 种指明目标

地址的方式，其基本含义类似存储器数据寻址的对应寻址方式，只是最终获得的是目标地址，如图 4-4 所示。

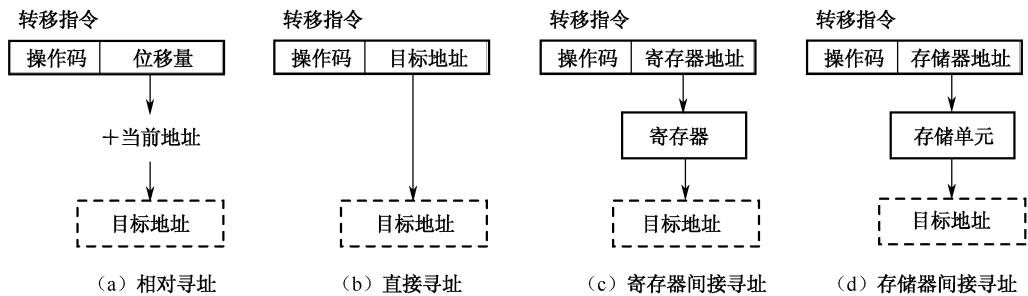


图 4-4 指令寻址方式

(1) 相对寻址方式

相对寻址是指指令代码提供目标地址相对于当前指令指针 IP 的位移量，转移后的目标地址（转移后的 IP 值）就是当前 IP 值加上位移量。由于要基于同一个基地址计算位置，所以相对寻址都是段内转移。

当同一个程序被操作系统安排到不同的存储区域执行时，指令间的位移并没有改变，不需改变转移地址。相对寻址方式给操作系统的灵活调度提供了很大的方便，是最常用的目标地址寻址方式。

(2) 直接寻址方式

直接寻址是由指令代码直接提供目标地址。8086 处理器只支持段间直接寻址。

(3) 间接寻址方式

间接寻址是指指令代码指示寄存器或存储单元，目标地址从寄存器或存储单元间接获得。如果用寄存器保存目标地址，称为目标地址的寄存器间接寻址；如果用存储单元保存目标地址，则称为目标地址的存储器间接寻址。

3. JMP 指令

JMP 指令被称为无条件转移（Jump）指令，就是无任何先决条件就能使程序改变执行顺序。处理器只要执行无条件转移指令 JMP，就可使程序转到指定的目标地址处，从目标地址处开始执行指令。JMP 指令相当于高级语言的 GOTO 语句。结构化的程序设计要求尽量避免使用 GOTO 语句，但指令系统决不能缺少 JMP 指令，汇编语言编程也不可避免地要使用 JMP 指令。

JMP 指令根据目标地址的转移范围和寻址方式，可以分成如下 4 种。

(1) 段内转移、相对寻址

```
jmp label ;IP=IP+位移量
```

段内相对转移 JMP 指令利用标号（Label）指明目标地址，常被采用。相对寻址的位移量是指，紧接着 JMP 指令后的那条指令的偏移地址到目标指令的偏移地址的地址位移。当向地址增大方向转移时，位移量为正；向地址减小方向转移时，位移量为负（补码表示）。由于是段内转移，只有 IP 指向的偏移地址改变，段寄存器 CS 内容不变。

(2) 段内转移、间接寻址

```
jmp r16 ;IP=r16, 寄存器间接寻址
jmp m16 ;IP=m16, 存储器间接寻址
```

段内间接转移 JMP 指令将一个 16 位通用寄存器或主存单元内容送入 IP 寄存器，作为新的指令指针，即偏移地址，但不修改 CS 寄存器的内容。

(3) 段间转移、直接寻址

jmp label ;IP=label 的偏移地址，CS=label 的段选择器

段间直接转移 JMP 指令是将标号所在的段选择器作为新的 CS 值，标号在该段内的偏移地址作为新的 IP 值。这样，程序跳转到新的代码段执行。

(4) 段间转移、间接寻址

jmp m32 ;IP=m32, CS=m32+2

段间间接转移 JMP 指令用一个双字存储单元表示要跳转的目标地址，将低字送 IP 寄存器，高字送 CS 寄存器（小端方式）。

像变量名一样，标号、段名、子程序名等标识符也具有地址和类型属性。所以，利用地址操作符 OFFSET 和 SEG，可以获得标号等的偏移地址和段地址。对应短转移、近转移和远转移依次类型名是 SHORT、NEAR 和 FAR，不同的类型汇编时将产生不同的指令代码。利用类型操作符 TYPE，可以获得标号等的类型值，如 NEAR 类型的标号返回 FF02H，FAR 类型的标号返回 FF05H。

MASM 汇编程序会根据存储模型和目标地址等信息自动识别是段内还是段间转移，也能够根据位移量大小自动形成短转移或近转移指令。同时，汇编程序提供了短转移 SHORT、近转移 NEAR PTR 和远转移 FAR PTR 操作符，强制转换一个标号、段名或子程序名的类型，形成相应的控制转移

【例 4-3】无条件转移程序。

			;数据段
0000	0000	nvar	dw ?
0002	00000000	fvar	dd ?
			;代码段
0017	EB 01	labl0:	jmp labl1 ;段内（短）转移、相对寻址
0019	90		nop
001A	E9 0001	labl1:	jmp near ptr labl2 ;段内（近）转移、相对寻址
001D	90		nop
001E	B8 0024 R	labl2:	mov ax,offset labl3
0021	FF E0		jmp ax ;段内转移、寄存器间接寻址
0023	90		nop
0024	B8 002F R	labl3:	mov ax,offset labl4
0027	A3 0000 R		mov nvar,ax
002A	FF 26 0000 R		jmp nvar ;段内转移、存储器间接寻址
002E	90		nop
002F	EA ---- 0035 R	labl4:	jmp far ptr labl5 ;段间转移、直接寻址
0034	90		nop
0035	B8 0047 R	labl5:	mov ax,offset labl6
0038	A3 0002 R		mov word ptr fvar,ax
003B	BA ---- R		mov dx,seg labl6
003E	89 16 0004 R		mov word ptr fvar+2,dx
0042	FF 2E 0002 R		jmp fvar ;段间转移、间接寻址
0046	90		nop
0047		labl6:	

本例主要用于理解指令寻址，左边罗列了列表文件内容，右边才是源程序本身。

本程序的第 1 条“`jmp lab11`”指令使处理器跳过一个空操作指令 NOP，执行标号 lab11 处的指令。由于 NOP 指令只有 1 字节，所以汇编程序将其作为一个相对寻址的短转移，其位移量用 1 字节表达为 01H。第 2 条 JMP 指令“`jmp near ptr lab12`”被强制生成相对寻址的近转移，因而其位移量用一个 16 位字表达，为 0001H。

指令“`jmp ax`”采用段内寄存器间接寻址转移到 AX 指向的位置，因为 AX 被赋值标号 lab13 的偏移地址，所以程序又跳过一个 NOP 指令，开始执行 lab13 处的指令。变量 nvar 保存了 lab14 的偏移地址，所以段内存储器间接寻址指令“`jmp nvar`”实现跳转到标号 lab14 处。

指令“`jmp far ptr lab15`”强制采用了段间直接寻址，转移到 lab15 标号。

双字变量 fvar 依次保存了标号 lab16 的偏移地址和段地址。所以指令“`jmp fvar`”控制程序流程转移到标号 lab16 处，它也是一个段间转移，但采用存储器间接寻址。

注意，fvar 不要定义为字变量，否则 MASM 会将其汇编成段内间接寻址。但是，如果 fvar 被定义为字变量，可以用 DWORD PTR 强制转换。另外，不需要使用 FAR PTR 操作符，否则被汇编成段间直接寻址。

JMP 指令既存在目标地址的寻址问题，同时存在数据的寻址问题，不要将二者混为一谈。例如，指令“`jmp nvar`”的指令寻址采用存储器间接寻址方式，而操作数 NVAR 的数据寻址则采用存储器直接寻址方式。存储器寻址方式有多种，所以该 JMP 指令的操作数还可以采用其他存储器寻址方式，如寄存器间接寻址：

```
mov bx,offset nvar
jmp near ptr [bx]
```

深入了解无条件转移指令的动态执行过程，可以参考例 4-1 的方法，将例 4-3 程序载入 DEBUG 调试程序，然后利用单步执行或断点执行的方法观察 JMP 指令的跳转情况，特别注意 IP 寄存器内容的变化。

4.2.2 条件转移指令

条件转移指令 JCC 根据指定的条件确定程序是否发生转移。如果满足条件，则程序转移到目标地址去执行程序；不满足条件，则程序将顺序执行下一条指令。其通用格式为：

```
jcc label           ;条件满足，发生转移，跳转到 LABEL 位置，即 IP=IP+位移量
                    ;否则，顺序执行
```

其中，label 表示目标地址，采用段内相对寻址方式。但应该注意，在 8086 处理器上，位移量只能用 1 字节表达，也就是只能实现 -128~+127 之间的短转移。

条件转移指令不影响标志，但要利用标志。条件转移指令 JCC 中的 CC 表示利用标志判断的条件，有 16 种，如表 4-2 所示。表中斜线分隔了同一条指令的多个助记符形式，目的是方便记忆。建议读者通过英文含义记忆助记符，掌握每个条件转移指令的成立条件。

根据判断的条件，条件转移指令可以分成两类。前 10 个为一类，它们将 5 个常用状态标志为 0 或 1 作为条件。后 8 个为另一类（其中有 2 个与前一类重叠），分别以两个无符号数据和有符号数据的 4 种大小关系为条件。

表 4-2 条件转移指令中的条件

助 记 符	标 志 位	英 文 含 义	中 文 说 明
jz/jje	ZF=1	Jump if Zero / Equal	等于零/相等
jnz/jjne	ZF=0	Jump if Not Zero / Not Equal	不等于零/不相等
js	SF=1	Jump if Sign	符号为负
jns	SF=0	Jump if Not Sign	符号为正
jp/jpe	PF=1	Jump if Parity/Parity Even	“1”的个数为偶
jnp/jpo	PF=0	Jump if Not Parity/Parity Odd	“1”的个数为奇
jo	OF=1	Jump if Overflow	溢出
jno	OF=0	Jump if Not Overflow	无溢出
jc/jb/jnae	CF=1	Jump if Carry / Below / Not Above or Equal	进位/低于/不高于等于
jnc/jnb/jae	CF=0	Jump if Not Carry / Not Below / Above or Equal	无进位/不低于/高于等于
jbe/jna	CF=1 或 ZF=1	Jump if Below or Equal / Not Above	低于等于/不高于
jnb/ja	CF=0 且 ZF=0	Jump if Not Below or Equal / Above	不低于等于/高于
jl/jnge	SF≠OF	Jump if Less / Not Greater or Equal	小于/不大于等于
jnl/jge	SF=OF	Jump if Not Less / Greater or Equal	不小于/大于等于
jle/jng	SF≠OF 或 ZF=1	Jump if Less or Equal / Not Greater	小于等于/不大于
jnl/jg	SF=OF 且 ZF=0	Jump if Not Less or Equal / Greater	不小于等于/大于

1. 单个标志状态作为条件的条件转移指令

这组指令单独判断 5 个状态标志之一，根据某个状态标志是 0 或 1 决定是否跳转：

- ⊙ JZ/JE 和 JNZ/JNE 利用零标志 ZF，分别判断结果是零（相等）还是非零（不等）。
- ⊙ JS 和 JNS 利用符号标志 SF，分别判断结果是负还是正。
- ⊙ JO 和 JNO 利用溢出标志 OF，分别判断结果是溢出还是没有溢出。
- ⊙ JP/JPE 和 JNP/JPO 利用奇偶标志 PF，判断结果低字节中“1”的个数是偶数还是奇数。
- ⊙ JC 和 JNC 利用进位标志 CF，判断结果是有进位（为 1）还是无进位（为 0）。

【例 4-4】 个数折半程序。

某个数组需要分成元素个数相当的两部分，所以需要对方个数进行折半。个数折半就是无符号整数除以 2。如果个数是一个偶数，商就是需要的半数；如果个数是奇数，余数为 1，商加 1 后作为半数。

无符号数除法运算可以使用除法指令 DIV，但使用逻辑右移指令 SHR 更加方便，被除数最低位就是余数，右移后进入 CF 标志。程序判断 CF 标志，CF=1 进行加 1 操作，CF=0 不需要再进行操作，直接获得结果。判断 CF 标志的指令是 JC 或 JNC。

```

;代码段
mov ax,885      ;假设一个数据
shr ax,1        ;数据右移进行折半
jnc goeven      ;余数为 0，即 CF=0 条件成立，不需要处理，转移
add ax,1        ;否则余数为 1，即 CF=1，进行加 1 操作
goeven:         call dispuiw    ;显示结果

```

为了观察到程序的运行结果，本例程序最后调用了十进制无符号数显示子程序 DISPUIW（参见附录 C），需要在源程序开始加入“INCLUDE IO.INC”语句。

本例程序使用了无进位（即余数为 0）转移指令 JNC，指令“add ax,1”是分支体。习惯了高级语言 IF 语句，也许会选择 JC 作为条件转移指令。程序片段如下：

```

mov ax,886      ;假设一个数据
shr ax,1        ;数据右移进行折半
jc goodd        ;余数为 1, 即 CF=1 条件成立, 转移到分支体, 进行加 1 操作
jmp goeven      ;余数为 0, 即 CF=0, 不需要处理, 转移到显示
goodd:          add ax,1      ;进行加 1 操作
goeven:         call dispuiw  ;显示结果

```

对比这两个程序片段, 显然后者多了一个 JMP 指令。可能读者会认为这个 JMP 指令是多余的。但如果没有这个 JMP 指令, 当个数是偶数时, JC 指令的条件不成立, 处理器将顺序执行下一条“add ax,1”指令, 则结果会被错误地多加 1。所以后一个程序片段看似符合逻辑, 但容易出错, 且多了一条跳转指令。

现代处理器中, 程序分支(或说条件转移指令)是影响程序性能的一个重要原因, 频繁的、复杂的分支会导致性能降低。程序员进行软件编程时可以运用一些编程技巧尽量避免分支。例如, 本例程序中可以用具有自动加 CF 的特点的 ADC 指令替代 ADD 指令, 从而避免使用条件转移指令:

```

mov ax,887      ;假设一个数据
shr ax,1        ;数据右移进行折半
adc ax,0         ;余数=CF=1, 进行加 1 操作; 余数=CF=0, 不需处理, 顺序执行
call dispuiw    ;显示结果

```

改进算法是提高性能的关键。例如, 不论个数是奇数还是偶数, 本例题都可以先将个数增 1, 然后除以 2 获得半数。这样避免了分支结构, 从而提高了性能。

```

mov ax,888      ;假设一个数据
add ax,1        ;个数加 1
rcr ax,1        ;数据右移进行折半
call dispuiw    ;显示结果

```

本程序片段采用 RCR 指令代替了 SHR 指令, 能正确处理 AX=FFFFH 时的特殊情况。因为 AX=FFFFH 加 1 后进位, AX=0; SHR 指令右移 AX 一位, AX=0; 而 RCR 指令带进位右移 AX 一位, AX=8000H; 显然后者结果正确。这就要求采用 ADD 指令实现加 1 影响进位标志, 而不能采用 INC 指令加 1 不影响进位标志。

【例 4-5】位测试程序。

进行底层程序设计时, 经常需要测试数据的某个位是 0 还是 1。例如进行打印前, 要测试打印机状态。假设测试数据已经进入 AL, 其 D1 位为 0 表示打印机没有处于联机打印的正常状态, D1 位为 1 表示可以进行打印。编程测试 AL, 若 D1=0, 显示“Not Ready!”; 若 D1=1, 显示“Ready to Go!”。

程序的主要问题是: 如何判断 AL 的 D1 位呢? 这个问题涉及数值中的某位, 可以考虑采用位操作类指令。例如, 用逻辑与将除 D1 位外的其他位变成 0, 保留 D1 位不变。测试 TEST 指令进行逻辑与 AND 操作, 且不改变操作数, 适合用于位测试。判断逻辑与运算后的这个数据是 0, 说明 D1=0; 否则, D1=1。判断运算结果是否为 0, 应该用零标志 ZF, 即应使用 JZ 或 JNZ 指令。

```

;数据段
no_msg db 'Not Ready!','$'
yes_msg db 'Ready to Go!','$'
;代码段
mov al,56h ;假设一个数据

```

```

test al,02h           ;测试 D1 位（使用 D1=1，其他位为 0 的数据）
jz nom               ;D1=0，条件成立，转移
mov dx,offset yes_msg ;D1=1，显示"Ready to Go!"
jmp done            ;跳转到另一个分支体
nom:                mov dx,offset no_msg ;显示"Not Ready!"
done:               mov ah,9
                   int 21h

```

请留意程序中的无条件转移 JMP 指令。该指令是必不可少的，因为若没有转移指令，程序将顺序执行，会在执行完一个分支后又进入另一个分支继续执行，产生错误。上述功能也可以使用不等于零转移指令 JNZ 实现，源程序如下：

```

mov al,58h           ;假设一个数据
test al,02h          ;测试 D1 位（使用 D1=1，其他位为 0 的数据）
jnz yesm            ;D1=1，条件成立，转移
mov dx,offset no_msg ;D1=0，显示"Not Ready!"
jmp done            ;跳转到另一个分支体
yesm:               mov dx,offset yes_msg ;显示"Ready to Go!"
done:               mov ah,9
                   int 21h

```

位测试还可以通过使用移位指令将要测试的位移进 CF 标志，然后通过 JC 或 JNC 指令进行判断来实现。

将例 4-5 载入调试程序可以获得更直观的感受，图 4-5 和图 4-6 演示了条件转移指令的调试过程，操作步骤详述如下：

```

-g 17
AX=0000 BX=0030 CX=0046 DX=00F5 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=0017  NO UP EI PL NZ NA PE NC
0DF3:0017 B056 MOV     AL,56
-t
AX=0056 BX=0030 CX=0046 DX=00F5 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=0019  NO UP EI PL NZ NA PE NC
0DF3:0019 A002 TEST    AL,02
-t
AX=0056 BX=0030 CX=0046 DX=00F5 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=001B  NO UP EI PL NZ NA PO NC
0DF3:001B 7405 JZ      0022
-t
AX=0056 BX=0030 CX=0046 DX=00F5 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=001D  NO UP EI PL NZ NA PO NC
0DF3:001D BA1900 MOV     DX,0019

```

图 4-5 条件转移指令的调试截图（1）

```

-t
AX=0056 BX=0030 CX=0046 DX=0019 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=0020  NO UP EI PL NZ NA PO NC
0DF3:0020 EB03 JMP     0025
-t
AX=0056 BX=0030 CX=0046 DX=0019 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=0025  NO UP EI PL NZ NA PO NC
0DF3:0025 B407 MOV     AH,07
-r ax
AX=0056
-t 19
AX=0058 BX=0030 CX=0046 DX=0019 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=001B  NO UP EI PL ZR NA PE NC
0DF3:001B 7405 JZ      0022
-t
AX=0058 BX=0030 CX=0046 DX=0019 SP=0430 BP=0000 SI=0000 DI=0000
DS=00F5 ES=0DE3 SS=0DF5 CS=0DF3 IP=0022  NO UP EI PL ZR NA PE NC
0DF3:0022 BA0E00 MOV     DX,000E

```

图 4-6 条件转移指令的调试截图（2）

（1）带被调试程序进入 DEBUG 调试程序，使用寄存器命令 R、反汇编命令 U 和显示命令 D 等了解被调试程序载入主存后的初始状态。

（2）使用“G 17”命令断点执行到完成语句“.startup”的指令，实现数据段和堆栈段设置。

(3) 单步执行到本例的测试指令 TEST, 观察零位标志 ZF 的当前状态。符号 NZ (No Zero, 见附录表 A-1) 表示结果不为零 (但 ZF=0)。下一条条件转移指令 JZ 判断的条件是结果等于零, 条件不成立, 所以不跳转, 顺序执行。这时, 继续单步执行, 可以观察到指令指针寄存器 IP 增量, 顺序执行, 进入本程序的 “显示'Ready to Go!'” 分支体。

(4) 如图 4-6 继续单步执行, 可以看到 JMP 指令实现跳转, 越过 “显示' Not Ready!'” 的另一个分支体。

(5) 用寄存器命令直接修改寄存器 AX 使 D1=0。这种情况下, 再次单步执行位于 0019H 地址的测试指令 TEST, 注意要使用等号指定执行指令的地址, 即 “T=19” 命令。

观察到零位标志 ZF 的状态是 ZR (Zero), 表示结果为 0 (但 ZF=1)。JZ 指令条件成立, 跳转到 “显示'Not Ready!'” 分支体。此时, 指令指针 IP 等于 0022H, 即 “JZ 0022” 指示的目标地址。

读者可以继续单步执行, 使用继续命令 P 显示信息, 还可以直接用寄存器命令 R 再次修改 ZF 标志状态, 重新执行分支程序, 观察标志对分支结构程序的影响, 体会条件的生成和判断。

【例 4-6】 奇校验程序。

数据通信时, 为了可靠常要进行校验, 最常用的校验方法是奇偶校验。如果使包括校验位在内的数据位为 “1” 的个数恒为奇数, 就是奇校验; 恒为偶数 (包括 0), 则为偶校验。例如, 标准 ASCII 码只有 7 位, 传输时可以再增加一个奇偶校验位 (作为最高位)。假设采用奇校验, 在字符 ASCII 码中为 “1” 的个数已为奇数时, 则令其校验位为 “0”, 否则令校验位为 “1”。奇偶校验标志 PF 正是为此而设计, 所以可以使用 JNP 或 JP 指令。

DOS 的 1 号功能调用实现从键盘输入一个字符, 它要求:

(1) 设置 AH 等于 1, 可以是指令:

```
mov ah,1
```

(2) 调用 DOS 功能, 即指令:

```
int 21h
```

调用 DOS 的 1 号功能, 系统等待用户输入, 待用户按键后完成调用, 在 AL 寄存器返回该键的 ASCII 码, 同时在屏幕上显示用户按下的字符。

编程为标准 ASCII 码最高位加上奇校验, 保存到 TDATA 变量中等待发送。

```

;数据段
Tdata    db ?           ;保存待发送数据的变量
;代码段
mov ah,1  ;1 号功能
int 21h   ;键盘输入
and al,7fh ;最高位置 “0”, 其他位不变, 同时标志 PF 反映 “1” 的个数
jnp next  ;个数为奇数, 则转向 NEXT
or al,80h  ;最高位置 “1”, 其他位不变
next:     mov Tdata,al ;保存待发送的数据
```

本例程序在判断出数据 “1” 的个数是奇数的情况下, 就不需执行任何指令, 只有为偶数 (包括 0) 个 “1” 时, 才需要执行最高位置 “1” 操作。

2. 以两数大小关系作为条件的条件转移指令

判断两个无符号数的大小关系和判断两个有符号数的大小关系要利用不同的标志位组

合，所以有对应的两组指令。

为了区别有符号数的大小关系，无符号数的大小关系用高（Above）、低（Below）表示，需要利用 CF 确定高低、利用 ZF 标志确定相等（Equal）。两个无符号数据的高低关系分成 4 种：低于（不高于等于）、不低于（高于等于）、低于等于（不高于）、不低于等于（高于）；依次对应 4 条指令：JB（JNAE），JNB（JAE），JBE（JNA），JNBE（JA）。

判断有符号数的大（Greater）、小（Less），需要组合 OF 和 SF 标志，并利用 ZF 标志确定相等与否。两个有符号数据的大小也分成 4 种关系：小于（不大于等于）、不小于（大于或等于）、小于等于（不大于）、不小于等于（大于）；也依次对应 4 条指令：JL（JNGE），JNL（JGE），JLE（JNG），JNLE（JG）。

两个数据还有是否相等的关系，这时不论是无符号数还是有符号数，都使用 JE 和 JNE 指令。相等的两个数据相减，结果当然为 0，所以 JE 等同于 JZ 指令；不相等的两个数据相减，结果一定不为 0，所以 JNE 等同于 JNZ 指令。

【例 4-7】 数据大小比较程序。

比较两个有符号数据之间的大小关系。如果两数相等，显示“Equal”；如果第 1 个数大，显示“First”；如果第 2 个数大，则显示“Second”。

```

;数据段
var1      dw -3765
var2      dw 8930
msg0      db 'Equal$'
msg1      db 'First$'
msg2      db 'Second$'

;代码段
mov ax,var1      ;取第 1 个数
cmp ax,var2      ;与第 2 个数比较
je equal         ;两数相等，转移
jnl first        ;第 1 个数大，转移
mov dx,offset msg2 ;第 2 个数大
jmp done
first:          mov dx,offset msg1
                jmp done
equal:          mov dx,offset msg0
done:          mov ah,9      ;显示结果
                int 21h
```

本例程序将数据作为有符号数，所以使用比较有符号数大小的条件转移指令 JNL。如果误用了比较无符号数大小的条件转移指令 JNB，程序运行的结果错误（结合补码表达，想想为什么）。

4.2.3 单分支结构

单分支程序结构是只有一个分支的程序，类似于高级语言的 IF-THEN 语句结构（没有 ELSE 语句）。前面例 4-4 和例 4-6 的分支程序就属于单分支结构。再如，计算有符号数据的绝对值，也是一个典型的单分支结构：正数无须处理，负数进行求补。

【例 4-8】 求绝对值程序。

```

;数据段
var      dw 0b422h      ;有符号数据
```



```

result    dw ?           ;保存绝对值
          ;代码段
          mov ax,var
          cmp ax,0        ;比较 AX 与 0
          jge nonneg      ;AX ≥ 0, 条件满足, 转移
          neg ax          ;AX < 0, 条件不满足, 为负数, 需求补得正值
nonneg:    mov result,ax   ;结束, 保存结果

```

单分支结构要注意采用正确的条件转移指令。条件满足（成立），则发生转移，跳过分支体；条件不满足，则顺序向下执行分支体，如图 4-7 所示。所以条件转移指令与高级语言的 IF 语句正好相反，IF 语句是条件成立时执行分支体。

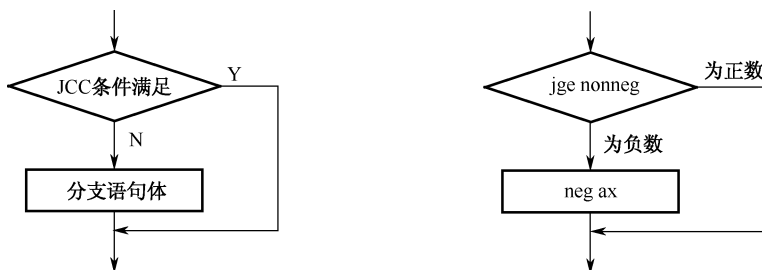


图 4-7 单分支结构的流程图

【例 4-9】 字母判断程序。

从键盘输入一个字符，判断是否为大写字母，为大写字母时，转换为小写并显示，为非大写字母，则退出。字符使用 ASCII 码，其值属于无符号数。

```

          ;代码段
          mov ah,1
          int 21h          ;输入一个字符, 从 AL 返回值
          cmp al,'A'       ;与大写字母 A 比较
          jb done          ;比大写字母 A 小, 不是大写字母, 转移
          cmp al,'Z'       ;与大写字母 Z 比较
          ja done          ;比大写字母 Z 大, 不是大写字母, 转移
          or al,20h        ;转换为小写
          mov dl,al
          mov ah,2
          int 21h          ;显示小写字母
done:

```

4.2.4 双分支结构

双分支程序结构有两个分支，条件为真执行一个分支，条件为假则执行另一个分支，相当于高级语言的 IF-THEN-ELSE 语句。如，前面例 4-5 和例 4-7 程序就属于双分支结构。再如，将数据最高位显示出来也可以采用双分支结构：最高位为 0 显示字符 0、为 1 显示字符 1。

【例 4-10】 显示数据最高位程序。

将最高位左移进入进位标志 CF，利用 JC（或者 JNC）判断出最高位是 1 还是 0，相应地显示“1”或者“0”。

```

;数据段
var      dw 0b422h      ;有符号数据
;代码段
mov bx,var
shl bx,1      ;BX 最高位移入 CF 标志
jc one       ;CF=1, 即最高位为 1, 转移
mov dl,'0'    ;CF=0, 即最高位为 0: DL←'0'
jmp two      ;一定要跳过另一个分支体
one:      mov dl,'1'    ;DL←'1'
two:      mov ah,2
int 21h      ;显示

```

双分支程序结构是条件满足发生转移执行分支体 2，条件不满足则顺序执行分支体 1；顺序执行的分支体 1 最后一定要有一条 **JMP** 指令跳过分支体 2，否则将进入分支体 2 而出现错误，如图 4-8 所示。**JMP** 指令必不可少，实现结束前一个分支回到共同的出口的作用。单分支结构中要选择跳过分支的转移条件，双分支结构选择条件转移指令可以比较随意，只要对应好分支体就可以了。

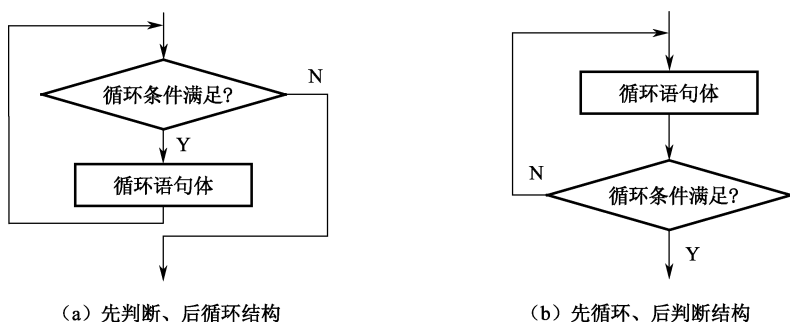


图 4-8 双分支结构的流程图

双分支结构有时可以改变为单分支结构，只要事先执行其中一个分支（选择出现概率较高的分支），当条件满足时就不再需要处理这个分支了。例 4-10 可修改为如下单分支结构：

```

;代码段
mov bx,var
mov dl,'0'      ;假设最高位为 0: DL←'0'
shl bx,1      ;BX 最高位移入 CF 标志
jnc two       ;CF=0, 即最高位为 0, 与假设相同, 转移
mov dl,'1'    ;CF=1, 即最高位为 1, DL←'1'
two:      mov ah,2
int 21h      ;显示

```

本例也可以使用 **ADC** 指令消除分支。

【例 4-11】 有符号数运算溢出程序。

虽然是同一个二进制编码，但作为有符号数和作为无符号数所表示的真值并不相同。在 8086 处理器指令系统中，乘法、除法和条件转移指令都针对有符号数和无符号数设计了两组不同的指令，但加法指令和减法指令对有符号数和无符号数却没有区别。加减法指令要求程序员利用溢出标志和进位标志区别对待有符号数和无符号数。具体地说，如果是两个有符号数进行加减，则应该防止其溢出（将数据位数扩大，使其能够表示结果，就不会出现溢出），

因为一旦溢出，运算结果就是错误的。如果是两个无符号数进行加减，则要利用进位或借位，因为虽然运算结果正确，但若有进位或借位，则运算结果必须包括进位或借位才是完整的。本例实现两个有符号数据相减，如果没有溢出，保存结果，显示正确信息；如果溢出，则显示错误信息。

```

;数据段
var1      dw 24680
var2      dw -9999
var3      dw ?
okmsg     db 'Correct!','$'      ;正确信息
errmsg     db 'ERROR ! Overflow!','$' ;错误信息
;代码段
mov ax,var1
sub ax,var2      ;求差
jo error        ;有溢出，转移
mov var3,ax      ;无溢出，保存差值
mov dx,offset okmsg ;显示正确
jmp disp
error:      mov dx,offset errmsg ;显示错误
disp:      mov ah,9
            int 21h

```

4.2.5 多分支结构

实际问题有时并不是单纯的单分支或双分支结构就可以解决，往往分支处理中又嵌套有分支，或者说具有多个分支走向，这就可以认为是逻辑上的多分支结构。一般利用单分支和双分支这两个基本结构，就可以解决程序中多个分支结构的问题。

例如，DOS 功能调用利用 AH 指定各个子功能，可以采用如下程序片段，实现多分支：

```

or ah,ah      ;等效于 CMP AH,0
jz function0  ;AH=0，转向 FUNCTION0
dec ah        ;等效于 CMP AH,1
jz function1  ;AH=1，转向 FUNCTION1
dec ah        ;等效于 CMP AH,2
jz function2  ;AH=2，转向 FUNCTION2
.....

```

典型的多分支结构类似于高级语言的 SWITCH 语句。汇编语言中常采用入口地址表的方法实现多分支，下面通过一个示例进行说明。

【例 4-12】 地址表程序。

假设有不超过 9 个信息（字符串）的数据表，编程显示指定的信息。具体功能如下：

- (1) 提示输入数字，并输入数字。
- (2) 判断数字是否在规定的范围内，若不在范围内，则重新输入。
- (3) 显示数字对应的信息，退出。

```

;数据段
msg1      db 'Chapter 1: Fundamentals',0dh,0ah','$'
msg2      db 'Chapter 2: Data Representation',0dh,0ah','$'

```

```

msg3      db 'Chapter 3: Basic Instructions',0dh,0ah,'$'
msg4      db 'Chapter 4: Program Structure',0dh,0ah,'$'
msg5      db 'Chapter 5: Procedure Programming',0dh,0ah,'$'
msg6      db 'Chapter 6: Windows Programming',0dh,0ah,'$'
msg7      db 'Chapter 7: 32-bit Instructions',0dh,0ah,'$'
msg8      db 'Chapter 8: Mixed Programming',0dh,0ah,'$'
msg9      db 'Chapter 9: Other Topics',0dh,0ah,'$'           ;9 个信息
msg       db 'Input number(1~9): $'           ;提示输入字符串
crlf      db 0dh,0ah,'$'           ;回车换行字符
table     dw disp1,disp2,disp3,disp4,disp5,disp6,disp7,disp8,disp9 ;地址表
;代码段
again:    mov dx,offset msg
          mov ah,9
          int 21h           ;提示输入
          mov ah,1          ;等待按键
          int 21h
          push ax            ;暂时将输入的按键字符保存到堆栈
          mov dx,offset crlf ;回车换行
          mov ah,9
          int 21h
          pop ax             ;恢复按键字符
          cmp al,'1'         ;数字 < 1?
          jb again
          cmp al,'9'         ;数字 > 9?
          ja again
          and ax,000fh       ;将 ASCII 码转换成数字
          dec ax
          shl ax,1           ;乘以 2，因为地址表是以 2 字节为单位
          mov bx,ax
          jmp table[bx]      ;（段内）间接转移：IP←[TABLE+BX]
          ;
disp1:    mov dx,offset msg1  ;分支程序 1
          jmp disp
disp2:    mov dx,offset msg2  ;分支程序 2
          jmp disp
disp3:    mov dx,offset msg3  ;分支程序 3
          jmp disp
disp4:    mov dx,offset msg4  ;分支程序 4
          jmp disp
disp5:    mov dx,offset msg5  ;分支程序 5
          jmp disp
disp6:    mov dx,offset msg6  ;分支程序 6
          jmp disp
disp7:    mov dx,offset msg7  ;分支程序 7
          jmp disp

```

```

disp8:    mov dx,offset msg8          ;分支程序 8
          jmp disp
disp9:    mov dx,offset msg9          ;分支程序 9
          jmp disp
          ;
disp:     mov ah,9                    ;显示
          int 21h

```

本例程序有 9 个分支，标号是 DISP1~DISP9。各分支程序都很简单，均为获得对应信息的存放地址，然后显示。为了实现分支，在数据段构造了一个地址表 TABLE，依次存放分支目标地址（使用标号表示其地址，也可以用 OFFSET 获得）。

输入正确的数字后，减 1 的目的是对应地址表，因为 1 号分支对应的 DISP1 标号地址存放在地址表位移量为 0 的位置。接着左移 1 位实现乘 2，因为分支地址是 16 位，在地址表中占 2 字节。例如，输入 3、减 1 为 2，乘 2 为 4，对应 DISP3 在地址表中的位移量也是 4。

利用地址表构造的多分支程序结构中，需要使用间接寻址的转移指令实现跳转。程序中“jmp table[bx]”指令的目标地址 IP 取自“TABLE+BX”指向的主存地址位置，正是对应的分支目标地址。

间接寻址的 JMP 转移指令还有其他形式，如示例程序的 JMP 指令还可以使用如下指令实现：

```

add bx,offset table      ;计算偏移地址
jmp word ptr [bx]        ;多分支跳转

```

针对本程序比较简单的功能，地址表中还可以直接存放信息字符串的地址，如下，可更简洁地完成要求：

```

          ;数据段
          .....          ;同上，略
table     dw msg1,msg2,msg3,msg4,msg5,msg6,msg7,msg8,msg9  ;地址表
          ;代码段
          .....          ;同上，略
          dec ax
          shl ax,1        ;乘以 2，因为地址表是以 2 字节为单位
          mov bx,ax
          mov dx,table[bx] ;获得信息字符串地址
          mov ah,9        ;显示
          int 21h

```

使用条件转移 JCC 指令和无条件转移 JMP 指令编写分支程序，是汇编语言的一个难点。条件转移指令并不支持一般的条件表达式，它是根据当前的某些标志位的设置情况实现转移或不转移。所以，必须根据实际问题将条件转换为标志或其组合，还要选择合适的指令产生这些标志。同时，还必须留心分支的开始点和结束点，当出现多分支时更应如此。

MASM 6.x 版本为了简化汇编语言的编程难度，引入了 .IF，.WHILE 等流程控制伪指令，使得汇编语言可以像高级语言那样编写分支和循环程序结构。读者在实际的程序开发中，可以利用这些高级语言的特性，本书将在第 7 章进行介绍。

4.3 循环程序结构

机器最适合完成重复性工作。程序设计中的许多问题都需要重复操作，如对字符串、数组等的操作。为了进行重复操作，不仅需要先做好准备，还要安排好退出的方法。完整的循环程序结构通常由 3 部分组成：

- ③ 循环初始部分——为开始循环准备必要的条件，如循环次数、循环体需要的初始值等。
- ③ 循环体部分——重复执行的程序代码，其中包括对循环条件的修改等。
- ③ 循环控制部分——判断循环条件是否成立，决定是否继续循环。

其中，循环控制部分是编程的关键和难点。循环控制可以在进入循环之前进行，则形成“先判断、后循环”的循环程序结构，对应高级语言的 WHILE 语句。如果循环之后进行循环条件判断，则形成“先循环、后判断”的循环程序结构，对应高级语言的 DO 语句，如图 4-9 所示。如果没有特殊情况，千万不要形成循环条件永远成立或无任何约束条件的死循环（永真循环、无条件循环）。

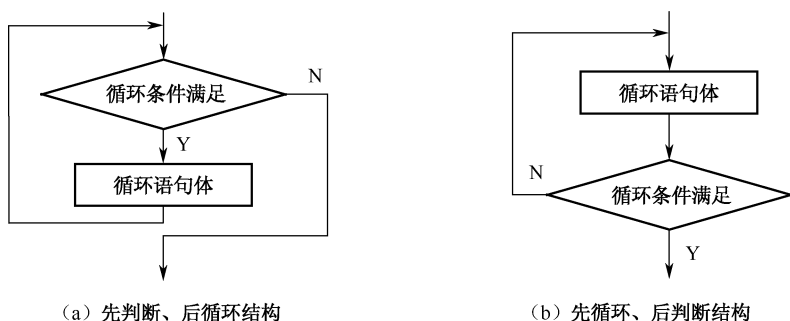


图 4-9 循环程序结构

8086 处理器有一组循环控制指令，用于实现简单的计数循环，即用于循环次数已知或者最大循环次数已知的循环控制。复杂的循环程序则需配合无条件和有条件转移指令才能实现。

4.3.1 循环指令

8086 处理器最主要的循环指令是 LOOP，在前面许多程序当中都使用了它。它使用 CX 寄存器作为计数器，每执行一次 LOOP 指令，CX 减 1（相当于指令“DEC CX”），然后判断 CX 是否为 0：如果不为 0，表示循环还没有结束，则转移到指定的标号处继续执行；如果为 0，则表示循环结束，顺序执行下条指令。后部分功能相当于不为 0 条件转移指令 JNZ。

循环指令 LOOP 的格式如下：

```
loop label          ;CX=CX-1
                    ;若 CX≠0，循环、跳转到 LABEL 位置，即：IP=IP+位移量
                    ;否则，顺序执行
```

还有 LOOPE / LOOPZ 和 LOOPNE / LOOPNZ 指令，它们在计数循环的基础上增加对 ZF 标志测试，即计数不归 0 并且结果是 0（LOOPE / LOOPZ 指令）或者计数不归 0 并且结果不是 0（LOOPNE / LOOPNZ 指令）才继续循环，否则顺序执行。

LOOP 指令的目标地址采用相对短转移，只能在-128~+127 字节之间循环。指令代码平均 3 字节，一个循环平均最多只能包含大约 42 条指令。有时常用 DEC 和 JNZ 指令组合实现，可以灵活利用其他寄存器作为计数器，不一定非得使用 CX 不可。

【例 4-13】 数组求和程序。

对一个数组中的所有元素进行求和，结果保存在变量中。假设数组元素是 16 位有符号整数，个数已知，运算过程中不考虑溢出问题。

对已知元素个数的数组进行操作，显然可以将个数作为计数值赋给 CX，控制循环次数；同时，需要一个通用寄存器作为元素的指针，并将求和的初值设置为 0。这些就是循环初始部分。循环体实现求和。计数循环的循环控制部分比较简单，就是将计数值减 1，不为 0 继续，对应 LOOP 指令。

```

;数据段
array      dw 136,-138,133,130,-161    ;数组
sum        dw ?                        ;结果变量

;代码段
xor ax,ax          ;求和初值为 0
mov cx,lengthof array ;CX=数组元素个数
mov bx,offset array ;BX=数组元素指针
again:  add ax,[bx]      ;求和
        add bx,type array ;指向下一个数组元素
        loop again
        mov sum,ax      ;保存结果

```

由于数组 ARRAY 是字量类型，所以“type array”等于 2，即每个数组元素占 2 字节，BX 加 2 指向下一个元素。

LOOP 指令先进行 CX 减 1 操作，然后进行判断。如果 CX 等于 0，则执行 LOOP 指令，共将循环 2¹⁶ 次。所以，如果数组元素的个数为 0，本程序将出错。为此，我们可以使用另一条循环指令 JCXZ 排除 CX 等于 0 的情况，该指令的格式为：

```

jcxz label      ;CX=0，转移、跳转到 LABEL 位置，即 IP=IP+位移量
                ;否则，顺序执行

```

在本例程序中，JCXZ 指令可以跟在设置 CX 的指令之后，跳转到保存结果。

图 4-10 和图 4-11 演示数组求和程序的调试过程，操作步骤如下。

(1) 带被调试程序启动 DEBUG 调试程序。使用寄存器命令 R、反汇编命令 U 等了解被调试程序载入主存后的初始状态。

```

D:\ML615>debug eg413.exe
-r
AX=0000  BX=0000  CX=003A  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000
DS=0DE3  ES=0DE3  SS=0DF7  CS=0DF3  IP=0000  NO UP EI PL NZ NA PO NC
0DF3:0000 BAF50D  MOV     DX,0DF5
u
0DF3:0000 BAF50D  MOV     DX,0DF5
0DF3:0003 8EDA      MOV     DS,DX
0DF3:0005 8CD3      MOV     BX,SS
0DF3:0007 2BDA      SUB     BX,DX
0DF3:0009 D1E3      SHL     BX,1
0DF3:000B D1E3      SHL     BX,1
0DF3:000D D1E3      SHL     BX,1
0DF3:000F D1E3      SHL     BX,1
0DF3:0011 FA        CLI
0DF3:0012 8ED2      MOV     SS,DX
0DF3:0014 03E3      ADD     SP,BX
0DF3:0016 FB        STI
0DF3:0017 3C09      XOR     AL,9
0DF3:0019 B90500     MOV     CX,0005
0DF3:001C BB0E00     MOV     BX,000E
0DF3:001F 0307      ADD     AX,[BX]

```

图 4-10 数组求和程序的调试截图 1

```

0017
0DF3:0017 33C0      XOR     AX,AX
0DF3:0019 B90500      MOV     CX,0005
0DF3:001C BB0E00      MOV     BX,000E
0DF3:001F 0307      ADD     AX,[BX]
0DF3:0021 03C302      ADD     BX,*02
0DF3:0024 E2F9      LOOP    001F
0DF3:0026 031900      MOV     [0019],AX
0DF3:0029 B44C      MOV     AH,4C
0DF3:002B CD21      INT     21
0DF3:002D 00880076    ADD     [BX+SI+7600],CL
0DF3:0031 FF850082    INC     WORD PTR [DI+8200]
0DF3:0035 005FFF      ADD     [BX-01],BL
-g 24
AX=0088 BX=0010 CX=0005 DX=0DF5 SP=0420 BP=0000 SI=0000 DI=0000
DS=0DF5 ES=0DE3 SS=0DF5 CS=0DF3 IP=0024  NO UP EI PL NZ AC PO NC
0DF3:0024 E2F9      LOOP    001F
-t
AX=0088 BX=0010 CX=0004 DX=0DF5 SP=0420 BP=0000 SI=0000 DI=0000
DS=0DF5 ES=0DE3 SS=0DF5 CS=0DF3 IP=001F  NO UP EI PL NZ AC PO NC
0DF3:001F 0307      ADD     AX,[BX]
DS:0010=FF76

```

图 4-11 数组求和程序的调试截图 2

通过前两条指令，可以观察到程序将设置数据段寄存器 DS 的段地址为 0DF5H；通过指令“MOV BX, 000E”，可判断出数组在数据段的偏移地址是 000EH。也许读者会奇怪数组变量的偏移地址为什么不是 0000H，这是因为简化段定义的 MASM 源程序格式下，代码段和数据段默认从模 2 地址（偶地址 xxx0B）开始，堆栈段默认从模 16 地址（可被 16 整除地址 xxxx0000B）开始。DOS 操作系统按照代码段、数据段、堆栈段顺序，将它们依次安排到主存，通常代码段和堆栈段的偏移地址是 0，但数据段的偏移地址不一定是 0。例如，在本程序中，代码段从 0DF3H:0000H 开始，结束于 0DF3H:002CH，下一个可用地址是 0DF3H:002DH（参看图 4-11 指令“INT 21”前的逻辑地址），即物理地址为 0DF5DH。这样，要求起始于偶地址的数据段只能从物理地址 0DF5EH 开始。由于段基址低 4 位必须是 0000B，故将物理地址 0DF5EH 低 4 位（十六进制 1 位）取 0 作为数据段高 16 位段地址是 0DF5H，则其偏移地址为 000EH。

（2）从偏移地址 0017H 开始反汇编，显示程序主体部分，包括循环程序，如图 4-11 所示。

（3）使用断点命令“G 24”从程序开始执行到循环指令 LOOP 前，循环体执行一次。观察到保存和值的寄存器 AX 等于数组第一个元素值，BX 等于 0010H（从 000EH 增加 2）指向第 2 个数组元素，而 CX 仍然等于 5，个数还没有减量。

（4）单步执行循环指令 LOOP，CX 减量成为 4，CX 不等于 0，所以继续循环，跳转到循环体开头，即 001FH 地址（对应源程序的 AGAIN 标号），也就是 IP=0017H。

（5）继续单步或断点执行，继续观察循环体执行情况，特别应留意 CX 的变化。最后，CX 等于 0，程序不再循环，顺序执行到保存和值。

4.3.2 计数控制循环

循环程序结构的关键是如何控制循环。比较简单的循环程序是通过次数控制循环，即计数控制循环。前面利用 LOOP 指令实现的程序都属于计数控制的循环程序。

【例 4-14】求最大值程序。

假设数组 ARRAY 由 16 位有符号整数组成，元素个数已知，没有排序。现要求编程获得其中的最大值。

求最大值（最小值）的基本方法就是逐个元素进行比较。由于数组元素个数已知，所以可以采用计数控制循环，每次循环完成一个元素的比较。循环体中包含分支程序结构。

```

;数据段
array      dw -3,0,20,900,587,-632,777,234,-34,-56    ;假设一个数组
count      = lengthof array                            ;数组的元素个数

```


max	dw ?	;存放最大值
	;代码段	
	mov cx,count-1	;元素个数减 1 是循环次数
	mov si,offset array	
	mov ax,[si]	;取出第一个元素给 AX, 用于暂存最大值
again:	add si,2	
	cmp ax,[si]	;与下一个数据比较
	jge next	;已经是较大值, 继续下一个循环比较
	mov ax,[si]	;AX 取得更大的数据
next:	loop again	;计数循环
	mov max,ax	;保存最大值

可以参考条件转移指令和数组求和程序的调试过程,将求最大值程序在 DEBUG 中运行,直观体会分支和循环相结合的程序调试。

【例 4-15】 简单加密解密程序。

逻辑异或 XOR 有一个特性: $X \oplus Y \oplus Y = X$, 即将一个数据 X 与另一个数据 Y 异或, 结果再与 Y 异或, 最后得到原来的 X (因为 $Y \oplus Y = 0$, 而 $X \oplus 0 = X$)。利用逻辑异或的这个特性, 可以实现简单的加密和解密。用户的明文逐个数据与密钥 Y 进行异或, 实现加密。加密后的密文再次与 Y 进行异或实现解密。

使用 1 字节量密钥, 将缓冲区的字符串加密保存, 可以显示加密后的密文。然后使用同一个密码进行解密, 并显示解密后的明文。

	;数据段	
key	db 234	;假设的一个密钥
buffer	db 'This is a secret.','\$'	;待加密的信息 (字符串)
count	= sizeof buffer-1	;不处理最后结尾字符
msg1	db 'Encrypted message: ','\$'	
msg2	db 13,10,'Original messge: ','\$'	
	;代码段	
	mov cx,count	;CX=字符个数, 作为循环的次数
	xor bx,bx	;BX 指向待处理的字符
	mov al,key	;AL=密钥
encrypt:	xor buffer[bx],al	;异或加密
	inc bx	;指向下一个字符
	cmp bx,cx	
	jb encrypt	;没有指向最后字符, 继续处理
	mov dx,offset msg1	;显示提示信息
	mov ah,9	
	int 21h	
	mov dx,offset buffer	;显示加密后的密文
	mov ah,9	
	int 21h	
	;	
	xor bx,bx	;BX 指向待处理的字符
	mov al,key	;AL=密钥
decrypt:	xor buffer[bx],al	;异或解密
	inc bx	
	dec cx	

```

jnz decrypt          ;等同于指令 loop decrypt
mov dx,offset msg2
mov ah,9
int 21h
mov dx,offset buffer ;显示解密后的明文
mov ah,9
int 21h

```

本例程序有两个雷同的循环程序部分，分别用于加密和解密。两次循环都利用字符个数作为控制循环条件，是计数控制循环结构。不过，循环加密时使用了增量方法，循环解密时像 LOOP 指令功能一样使用减量方法。

本程序的加解密算法都很简单，并且使用了事先设置的密钥，很容易被攻破。

4.3.3 条件控制循环

复杂的循环程序结构需要利用条件转移指令，根据条件决定是否进行循环，这就是所谓的条件控制循环。计数控制循环往往至少执行一次循环体之后，才判断次数是否为 0，这是所谓的“先循环、后判断”循环结构。条件控制循环更多见的是“先判断、后循环”结构。

【例 4-16】 字符个数统计程序。

已知某个字符串以 0 结尾，统计其包含的字符个数，即计算字符串长度。这是一个循环次数不定的循环程序结构，宜用转移指令决定是否结束循环，并应该先判断、后循环。循环体仅进行简单的个数加 1 操作。

```

;数据段
string    db 'Do you have fun with Assembly?',0    ;以 0 结尾的字符串
;代码段
xor bx,bx;BX 用于记录字符个数，同时也用于指向字符的指针
again:    mov al,string[bx]
          cmp al,0                                ;也可以使用指令 “test al,al”
          jz done
          inc bx                                  ;个数加 1
          jmp again                               ;继续循环
done:     mov ax,bx                               ;显示个数
          call dispuiw

```

先行判断的条件控制循环程序很像双分支结构，只不过一个主要分支需要重复执行多次（所以跳转指令 JMP 的目标位置是循环开始，而不是跳过另一个分支，到达双分支的汇合地），另一个分支则用于跳出循环。先行循环的条件控制循环程序类似于单分支结构，循环体就是分支体，顺序执行就跳出循环。

为了显示统计的字符个数，最后调用了本书提供的子程序 DISPUIW（参见附录 C）。

计算机中表达字符串时常用 3 种方法标识结束。其中，固定长度的方法最简单，但不够灵活。例如，在 Pascal 等语言中，字符串最开始的单元存放该字符串的长度。比较常用的方法是使用结尾字符，也就是字符串最后使用一个特殊的标识符号。结尾字符曾使用过字符“\$”（如 DOS 的 9 号功能调用）、回车字符 CR（ASCII 值是 13）、换行字符 LF（ASCII 值是 10）等，现在多使用 0（即 ASCII 表的第一个字符，常表达为 NULL 或 NUL 常量）。使用 0 作为字符串结尾，是 C/C++ 和 Java 语言的规定，也可以避免在字符串中出现结尾字符的情况，应

该说是比较理想的方法。

【例 4-17】 斐波那契数列程序。

斐波那契 (Fibonacci) 数列 (1, 1, 2, 3, 5, 8, 13, ...) 是用递推方法生成的一系列自然数:

$$F(1)=1$$

$$F(2)=1$$

$$F(N)=F(N-1)+F(N-2); N\geq 3$$

即遵循“从第 3 个数开始, 每一个数为前两个数的和”的规律生成的数列。编程输出斐波那契数, 一行一个, 直到超出 16 位数据范围, 不能表达为止。

```
      ;代码段
      mov ax,1      ;AX=F(1)=1
      call dispuiw  ;显示第 1 个数
      call dispCrLf ;回车换行
      call dispuiw  ;显示第 2 个数
      call dispCrLf ;回车换行
      mov bx,ax     ;BX=F(2)=1
again: add ax,bx     ;AX=F(N)=F(N-2)+F(N-1)
      jc done
      call dispuiw  ;显示一个数
      call dispCrLf ;回车换行
      xchg ax,bx    ;AX=F(N-2), BX=F(N-1)
      jmp again

done:
```

寄存器保存的是 16 位无符号整数, 所以这里的超出范围是出现进位 (不是有符号整数的溢出), 需要利用条件转移指令 JC 退出循环。

为了能够在屏幕上显示十进制形式的斐波那契数列, 本例程序不仅使用了本书提供的子程序 DISPUIW, 还使用了子程序 DISPCRLF 实现光标回车换行, 即回到下行首列位置。

想保存输出的斐波那契数列吗? 可以使用 DOS 支持的重定向功能, 即符号 “>”。假设本例程序的可执行文件是 EG417A.EXE, 要将斐波那契数列保存在 EG417.TXT 文件中, 在命令行输入如下命令即可:

```
eg417a.exe > eg417.txt
```

4.3.4 多重循环

实际的应用问题不会只有单纯的分支或循环, 两者可能同时存在, 即循环体中具有分支结构, 分支体中采用循环结构。有时, 循环体中嵌套有循环, 即形成多重循环结构。在多重循环中, 如果内外循环之间没有关系, 则问题比较容易处理; 但如果需要传递参数或利用相同的数据, 问题就比较复杂了。

【例 4-18】 冒泡法排序程序。

实际的排序算法很多, “冒泡法” 是一种易于理解和实现的方法, 但并不是最优的算法。冒泡法从第一个元素开始, 依次对相邻的两个元素进行比较, 使前一个元素不大于后一个元素; 将所有元素比较完之后, 最大的元素排到了最后; 然后, 除最后一个元素之外的元素依上述方法再进行比较, 得到次大的元素排在后面; 如此重复, 直至完成, 以实现元素从小到大

大的排序，如图 4-12 所示。可见，这是一个双重循环程序结构。外循环由于循环次数已知，可用 LOOP 指令实现；而内循环次数每次外循环后减少一次，用 DX 表示。循环体比较两个元素大小，又是一个分支结构。

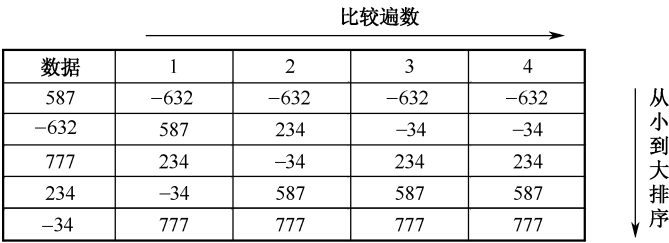


图 4-12 冒泡法的排序过程

```

;数据段
array    dw 587,-632,777,234,-34    ;假设一个数组
count    = lengthof array           ;数组的元素个数
;代码段
mov cx,count                         ;CX←数组元素个数
dec cx                               ;元素个数减 1 为外循环次数
outlp:   mov dx,cx                   ;DX←内循环次数
         mov bx,offset array
inlp:    mov ax,[bx]                 ;取前一个元素
         cmp ax,[bx+1]               ;与后一个元素比较
         jng next                    ;前一个不大于后一个元素，则不进行交换
         xchg ax,[bx+1]              ;否则，进行交换
         mov [bx],ax
next:    inc bx                      ;下一对元素
         dec dx
         jnz inlp                    ;内循环尾
         loop outlp                  ;外循环尾

```

【例 4-19】 字符剔除程序。

现有一个以 “\$” 结尾的字符串，要求剔除其中的空格字符。

本例程序可以用结尾标志 “\$” 作为循环控制条件。循环体判断每个字符，如果不是空格，则不予处理继续循环；若是空格，则进行剔除，也就是将后续所有字符逐个前移一个字符位置，将空格覆盖。这是一个 “先判断、后循环” 结构。

因为有多个字符需要前移，所以又需要一个循环，循环结束条件仍然使用结尾标志 “\$”。这个循环则是 “先循环、后判断” 结构。最终形成一个双重循环程序结构。

```

;数据段
string   db 'Let us have a try !',0dh,0ah,'$'    ;以 “$” 结尾的字符串
;代码段
mov dx,offset string    ;显示处理前的字符串
mov ah,9
int 21h
mov si,offset string
outlp:   cmp byte ptr [si],'$'                    ;外循环，先判断后循环

```

	jmp done	;为“\$”，则结束
again:	cmp byte ptr [si],'	;检测是否是空格
	jnz next	;不是空格，继续循环
	mov di,si	;是空格，进入剔除空格分支
inlp:	inc di	;该分支是循环程序
	mov al,[di]	;前移一个位置
	mov [di-1],al	
	cmp byte ptr [di],'\$'	;内循环，先循环后判断
	jnz inlp	;内循环结束处
	jmp again	;再次判断是否为空格（处理连续空格情况）
next:	inc si	;继续对后续字符进行判断处理
	jmp outlp	;外循环结束处
done:	mov dx,offset string	;显示处理后的字符串
	mov ah,9	
	int 21h	

本程序采用的剔除算法并不优秀。如果已知字符串的长度，更好的算法应该从字符串最后一个字符开始判断处理，这样可以减少一些移动次数。因为从前面开始移动，后续的空格也要随之移动；而如果从后面开始移动，就不会因空格进行无谓的移动了。读者可自行练习。

4.3.5 串操作指令

以字节、字和双字为单位的多个数据存放在连续的主存区域中就形成数据串（String），也就是数组（Array），如以字节为单位的 ASCII 码字符串就是典型的数据串。数据串是程序经常需要处理的数据结构，前面计算字符串长度、数组排序等循环结构程序都是串操作。为了方便地进行数据串操作，8086 处理器特别为此设计了串操作类指令，很有特色。

根据串数据类型的特点，串操作指令采用了特殊的寻址方式：

- ⊙ 源操作数用寄存器 SI 间接寻址，默认在数据段 DS 中，即 DS:[SI]，允许段超越。
- ⊙ 目的操作数用寄存器 DI 间接寻址，默认在附加段 ES 中，即 ES:[DI]，不允许段超越。
- ⊙ 每执行一次串操作，源指针 SI 和目的指针 DI 将自动修改±1 或±2。
- ⊙ 对于以字节为单位的数据串（指令助记符用 B 结尾）操作，地址指针应该为±1。
- ⊙ 对于以字为单位的数据串（指令助记符用 W 结尾）操作，地址指针应该为±2。
- ⊙ 当方向标志 DF=0（执行 CLD 指令设置），地址指针应该为+1 或+2。
- ⊙ 当方向标志 DF=1（执行 STD 指令设置），地址指针应该为-1 或-2。

串操作后之所以自动修改 SI 和 DI 指针，是为了方便对后续数据进行操作，修改的数值对应数据串单位所包含的字节数。用户通过执行 CLD 或 STD 指令控制方向标志 DF，决定主存地址是增大（DF=0，向地址高端增量）还是减小（DF=1，向地址低端减量）。

串操作指令有两组：一组实现数据串传送，另一组实现数据串检测。串操作通常需要重复进行，所以经常配合重复前缀指令，它通过计数器 CX 控制重复执行串操作指令的次数。

1. 串传送指令

这组串操作指令实现对数据串的传送（MOVS）、存储（STOS）和读取（LODS），可以配合 REP 重复前缀，它们不影响标志。

（1）串传送指令 MOVS 将数据段中的字节或字数据，传送至 ES 指向的段：

```
movsb    ;字节串传送: ES:[DI]←DS:[SI]; 然后: SI←SI±1, DI←DI±1
movsw    ;字串传送: ES:[DI]←DS:[SI]; 然后: SI←SI±2, DI←DI±2
```

(2) 串存储指令 STOS 将 AL 或 AX 内容存入 ES 指向的段:

```
stosb    ;字节串存储: ES:[DI]←AL; 然后: DI←DI±1
stosw    ;字串存储: ES:[DI]←AX; 然后: DI←DI±2
```

(3) 串读取指令 LODS 将数据段中的字节或字数据读到 AL 或 AX:

```
lodsb    ;字节串读取: AL←DS:[SI]; 然后: SI←SI±1
lodsw    ;字串读取: AX←DS:[SI]; 然后: SI←SI±2
```

(4) 重复前缀指令 REP 用在 MOVS、STOS 和 LODS 指令前, 利用计数器 CX 保存数据串长度, 可以理解为“当数据串没有结束 (CX≠0), 则继续传送”:

```
rep      ;每执行一次串指令, CX 减 1; 直到 CX=0, 重复执行结束
```

需要注意, 串操作指令本身仅进行一个数据的操作, 利用重复前缀才能实现连续操作。重复前缀指令先判断 CX 是否为 0, 为 0 结束; 否则进行减 1 操作, 并执行串操作指令。

【例 4-20】 字符串复制程序。

例 2-11 程序实现字符串复制是串传送指令 MOVS 的主要应用情况。

```
          ;数据段
srcmsg    db 'Try your best, why not.$'
dstmsg    db sizeof srcmsg dup(?)

          ;代码段
mov ax,ds
mov es,ax          ;设置附加段 ES=DS
mov si,offset srcmsg ;SI=源字符串地址
mov di,offset dstmsg ;DI=目的字符串地址
mov cx,lengthof srcmsg ;CX=字符串长度
cld              ;地址增量传送
rep movsb        ;重复进行字符串传送
mov ah,9         ;显示字符串
mov dx,offset dstmsg
int 21h
```

本例程序将源字符串 SRCMSG 内容复制到目的字符串 DESTMSG 中。使用串传送指令 MOVS, 需要事前设置 DS、ES、SI、DI 和方向标志 DF, 并将 CX 赋值需要重复的次数。这样, 简单的一条指令就完成了全部的传送工作。如果不使用重复前缀, 则需要使用循环指令:

```
again:    movsb
          loop again
```

当然, 本例程序也可以不使用数据串指令, 参见例 2-11 和例 2-12。

MOVS 指令每次只传送 1 字节数据, 若字符串很长, 可使用 MOVSW 来提高效率:

```
shr cx,1          ;长度除以 2, 最低位进入 CF
rep movsw         ;以字为单位重复传送, 最后 CX=0
rcl cx,1          ;保存在 CF 的原 CX 最低位, 重新移入 CX (剩余个数: 0 或 1)
rep movsb        ;以字节为单位传送剩余的字符
```

【例 4-21】 直接清除屏幕程序。

串存储指令 STOS 经常用于填充数据。

在 DOS 的标准显示模式下, 屏幕由 25 行、每行 80 列字符组成 (25×80 显示模式)。每个字符由 2 字节控制显示, 高字节为字符属性字节, 如 07H 是标准的黑底白字; 低字节为字

符的 ASCII 码。从逻辑地址 B800H:0000H 开始的显示缓冲区，每个字单元内容对应一个显示字符，共 25×80 个字单元。

本例程序将 B800H:0000H 开始的 25×80 个字单元全部填入 0720H，实现清除屏幕的目的（相当于 DOS 的清屏命令 CLS）。

```
;代码段
mov dx,0b800h
mov es,dx
mov di,0           ;设置 ES: DI=B800H:0000H
mov cx,25*80       ;设置 CX=填充个数
mov ax,0720h       ;设置 AX=填充内容
cld
rep stosw
```

本例程序中设置 AL 为 20H，即空格字符，将显示缓冲区填充为空格，实现了清除屏幕作用（程序运行后，最后显示的命令行是 DOS 加上的）。如果将 AX 值设置为 0731H，则屏幕将充满数字 1；如果 AX=0141H，则屏幕将充满蓝色字母 A。

由于本例程序默认采用 DOS 标准显示模式，所以只能在 DOS 环境的标准显示模式下运行，而不能使用 32 位控制台窗口（CMD.EXE）。

本例程序没有通过功能调用，而是直接读写显示缓冲区来实现显示输出，故常被称为“直接写屏”方法。这是直接针对硬件操作的 I/O 程序，属于底层的驱动程序。

2. 串检测指令

串检测指令实现对数据串的比较（CMPS）和扫描（SCAS）。由于串比较和扫描的实质是进行减法运算，所以它们像减法指令一样影响标志位。这两个串操作指令可以配合重复前缀 REPE/REPZ 和 REPNE/REPNZ，通过 ZF 标志说明两数是否相等。

（1）串比较指令 CMPS 用源数据串减去目的数据串，以比较两者间的关系：

```
cmpsb    ;字节串比较：DS:[SI]-ES:[DI]；然后：SI←SI±1，DI←DI±1
cmpsw    ;字串比较：DS:[SI]-ES:[DI]；然后：SI←SI±2，DI←DI±2
```

注意：串比较指令 CMPS 是源操作数（SI 指向的主存数据）减去目的操作数（DI 指向的主存数据），而比较指令 CMP 是目的操作数减去源操作数。

（2）串扫描指令 SCAS 用 AL 或 AX 内容减去目的数据串，以比较两者间的关系：

```
scasb    ;字节串扫描：AL-ES:[DI]；然后：DI←DI±1
scasw    ;字串扫描：AX-ES:[DI]；然后：DI←DI±2
```

（3）重复前缀指令 REPE（或 REPZ）用在 CMPS 和 SCAS 指令前，利用计数器 CX 保存数据串长度，同时判断比较是否相等，可以理解为“当数据串没有结束（CX≠0），并且串相等（ZF=1），则继续比较”：

```
repe|repz ;每执行一次串指令，CX 减 1；只要 CX=0 或 ZF=0，重复执行结束
```

（4）重复前缀指令 REPNE（或 REPNZ）也用在 CMPS 和 SCAS 指令前，利用计数器 CX 保存数据串长度，同时判断比较是否不相等，可以理解为“当数据串没有结束（CX≠0），并且串不相等（ZF=0），则继续比较”：

```
repne|repne ;每执行一次串指令，CX 减 1；只要 CX=0 或 ZF=1，重复执行结束
```

重复执行结束的条件是“或”的关系，只要满足条件之一即可。所以指令执行完成，可能数据串没有比较完，也可能数据串已经比较完，编程时需要区分。

重复前缀指令先判断 CX 是否为 0，为 0 结束（如果初始化 CX 为 0，将不会重复操作）；否则进行减 1 操作，并执行串操作指令（LOOP 指令是先减 1 后判断是否为 0）；最后判断 ZF 标志是否符合继续循环的条件。图 4-13 总结了串操作的过程。

【例 4-22】 等长字符串比较程序。

```

;数据段
string1 db 'equal or not'
string2 db 'eQual or not'
;代码段
mov ax,ds
mov es,ax      ;设置附加段 ES=DS
mov cx,sizeof string1
mov si,offset string1
mov di,offset string2
cld
repz cmpsb     ;重复比较,不同或比较完结束比较
jne found      ;发现不同字符, 转移到 FOUND
mov dl,'Y'     ;字符串相同, 显示 Y
jmp done
found: mov dl,'N' ;字符串不同, 显示 N
done:  mov ah,2
       int 21h

```

本例程序比较两个长度相等的字符串，如果两个字符串相同，则显示 Y，否则显示 N。

指令“repz cmpsb”结束重复执行的情况有两种：第一种是出现不相等的字符（ZF=0），第二种是比较完所有字符（CX=0）。在第二种情况下，对最后比较的一对字符又有两种可能：最后字符不相等（ZF=0），或者最后字符相等（ZF=1），也就是两个字符串相同。所以，重复比较结束后，指令 JNE 的条件成立（ZF=0）表示字符串不相等。

本例字符串长度为 12，比较的两个字符串第 1 个字符相同，第 2 个字符不相同，根据串操作流程图 4-13，结束重复执行时 CX=10。

【例 4-23】 字符串查找程序。

编程应用中经常需要查找某个特定数据，使用串扫描 SCAS 很方便。

本例程序在 DOS 管理的最低 32KB 主存区域内搜索字符串“COMMAND”，因为 DOS 环境下确实存在这个字符串，所以程序显示“Y”。

```

;数据段
search db 'COMMAND' ;待查找的字符串
;代码段
mov dx,0
mov es,dx
mov di,dx          ;逻辑地址 (ES:DI=0:0) 起始
mov cx,8000h       ;32KB 主存空间
cld

```

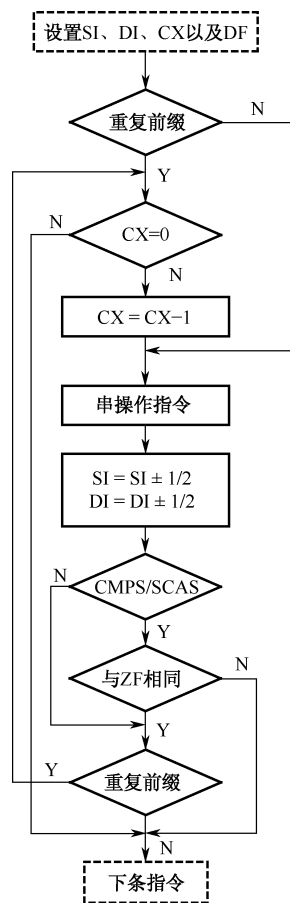


图 4-13 重复串操作的流程图


```

again1:  mov dx,sizeof search ;DX=待查找字符串长度
         mov si,offset search ;SI=待查找字符串地址
         lodsb                ;取出第一个待比较的字符 AL=DS:[SI], SI=SI+1
         repnz scasb          ;重复扫描: AL=ES:[DI], DI=DI+1
         jcxz next            ;CX=0, 扫描到最后字符, 转移
         mov bx,di            ;不是最后, 保存第一个字符相同时的地址

again2:  dec dx
         jz found              ;比较完所有字符, 查找到
         lodsb                ;比较下一个字符
         scasb
         jz again2            ;还相同, 继续比较
         mov di,bx             ;不是完全相同, 恢复第一个字符相同时的地址
         jmp again1           ;重新从第一个字符开始比较

next:    jnz nofound           ;最后一个字符不相同, 没有查找到
         dec dx
         jz found              ;待查找内容只有一个字符, 查找到

nofound: mov dl,'N'            ;未查找到, 显示 N
         jmp done

found:   mov dl,'Y'            ;查找到, 显示 Y
done:    mov ah,2
         int 21h

```

字符串查找需要逐个字符比较，其中关键是比较第一个字符。第一个字符不相同，需要重复进行；只有在第一个字符相同的情况下，再继续比较下面的字符；而只要有一个字符不相同，就需要重新从下一个主存单元开始查找。在后续的比较过程中，需要暂存第一个字符相同时的地址，以便不完全相同时从这个位置开始下一轮比较。

指令“`repnz scasb`”停止重复扫描还可能是比较完所有的主存单元，但需要判断最后一个字符是否相同。在相同的情况下，只有比较完字符串中所有字符（也有可能字符串只有一个字符的特例）才能确定是完全相同，查找到需要的字符串。

本例程序是多重循环，内、外循环都要使用主存地址（程序中使用 `DI` 保存），要注意保存。本例程序也可以使用串比较 `CMPS` 指令来实现，更简捷。当然，可以不使用串操作指令实现编程要求，也不再受必须使用 `SI` 和 `DI` 的限制，更灵活。核心的双重循环部分如下：

```

again1:  xor bx,bx              ;使用 BX 指示逐个字符比较
again2:  mov al,search[bx]
         cmp al,[bx+di]         ;比较一个字符
         jne next
         inc bx                 ;指向下一个字符
         cmp bx,sizeof search
         jz found               ;字符串比较完, 仍相等, 则查找到
         jmp again2

next:    inc di                 ;有不不同的字符
         loop again1            ;重新从第一个字符开始比较
         mov dl,'N'             ;未查找到, 显示 N
         jmp done

found:   mov dl,'Y'             ;查找到, 显示 Y
done:    mov ah,2
         int 21h

```

很多时候不采用串操作指令也很方便，简单指令组成的程序代码往往更高效。

习 题 4

4.1 简答题

- (1) 从 CMOS RAM 的 9、8 和 7 号单元均读出 08H，表示哪年哪月哪日？
- (2) 数据的直接寻址和指令的直接寻址有什么区别？
- (3) 是什么特点决定了目标地址的相对寻址方式应用最多？
- (4) JCC 指令能跳转到代码段之外吗？
- (5) 什么是奇偶校验？
- (6) 助记符 JZ 和 JE 为什么表达同一条指令？
- (7) 为什么判断无符号数大小和有符号大小的条件转移指令不同？
- (8) 双分支结构中两个分支体之间的 JMP 指令有什么作用？
- (9) 如果循环体的代码量远超过 128 字节，还能用 LOOP 指令实现计数控制循环吗？
- (10) 什么是“先循环、后判断”循环结构？

4.2 判断题

- (1) 指令指针或者还包括代码段寄存器值的改变将引起程序流程的改变。
- (2) 指令的相对寻址都是段内转移。
- (3) 采用指令的寄存器间接寻址，目标地址来自存储单元。
- (4) JMP 指令对应高级语言的 GOTO 语句，所以不能使用。
- (5) 因为条件转移指令 JCC 要利用标志作为条件，所以也影响标志。
- (6) JA 和 JG 指令的条件都是“大于”，所以是同一个指令的两个助记符。
- (7) JC 和 JB 指令的判断条件都是 $CF=1$ ，所以是同一条指令。
- (8) 控制循环是否结束只能在一次循环结束之后进行。
- (9) 介绍 LOOP 指令时常说它相当于 DEC CX 和 JNZ 两条指令。但考虑对状态标志的影响，它们有差别。LOOP 指令不影响标志，DEC 指令却会影响除 CF 之外的其他状态标志。
- (10) 若 $CX=0$ ，则 LOOP 指令和 JCX 指令都发生转移。

4.3 填空题

- (1) JMP 指令根据目标地址的转移范围和寻址方式，可以分成 4 种类型：段内转移、____，段内转移、____，段间转移、____，段间转移、____。
- (2) MASM 给短转移、近转移和远转移定义的类型名依次是____、____和____。
- (3) 假设 $BX=1256H$ ，字变量 TABLE 的偏移地址是 $20A1H$ ，数据段偏移地址 $32F7H$ 处存放 $3280H$ ，执行指令“JMP BX”后 $IP=$ ____，执行指令“JMP TABLE[BX]”后 $IP=$ ____。
- (4) “CMP AX,3721H”指令之后是 JZ 指令，发生转移的条件是 $AX=$ ____，此时 $ZF=$ ____。
- (5) 执行“SHR BX,1”指令后，JNC 发生转移，说明 BX 的 $D0=$ ____。
- (6) 在 DX 等于 0 时转移，可以使用指令“CMP DX,____”，也可以使用“TEST DX,____”构成条件，然后使用 JE 指令实现转移。
- (7) 循环结构程序一般有 3 个部分组成，它们是____，循环体和____部分。
- (8) JCXZ 指令发生转移的条件是____，LOOP 指令不发生转移的条件是____。

(9) LOOP 指令进行减 1 计数, 实际应用中常进行加 1 计数。针对例 4-13 程序, 如果删除其中的 LOOP 指令, 则可以使用指令“CMP ____, CX”和“JB ____”替代。

(10) 小写字母“e”是英文当中出现频率最高的字母。如果某个英文文档利用例 4-15 的异或方法进行简单加密, 统计发现密文中字节数据“8FH”最多, 则该程序采用的字节密码可能是____。

4.4 已知 var1、var2、var3 和 var4 是 16 位无符号整数, 用汇编语言程序片段实现如下 C 语句:

```
var4=(var1*6)/(var2-7)+var3
```

4.5 已知 var1、var2、var3 和 var4 是 16 位有符号整数, 用汇编语言程序片段实现如下 C 语句:

```
var1=(var2*var3)/(var4+8)-47
```

4.6 参看例 4-1, 假设 N 小于 9000, 这时求和结果只需要 AX 保存, DX 为 0。修改例 4-1 使其可以从键盘输入一个数值 N (用 READUIW 子程序), 最后显示累加和(用 DISPUIW 子程序)。

4.7 为了验证例 4-3 程序的执行路径, 可以在每个标号前后增加显示一个数字的功能(利用 2 号 DOS 功能), 使得程序运行后显示数码 123456。

4.8 执行如下程序片段后, CMP 指令分别使得 5 个状态标志 CF、ZF、SF、OF 和 PF 为 0 还是为 1? 它会使得哪些条件转移指令指令 JCC 的条件成立, 发生转移?

```
mov ax,20h
cmp ax,80h
```

4.9 判断下列程序段跳转的条件。

(1) xor ax,1e1eh

```
je equal
```

(2) test al,10000001b

```
jnz here
```

(3) cmp cx,64h

```
jb there
```

4.10 假设 BX 和 SI 存放的是有符号数, DX 和 DI 存放的是无符号数, 请用比较指令和条件转移指令实现以下判断:

(1) 若 $DX > DI$, 转到 above 执行;

(2) 若 $BX > SI$, 转到 greater 执行;

(3) 若 $BX = 0$, 转到 zero 执行;

(4) 若 $BX - SI$ 产生溢出, 转到 overflow 执行;

(5) 若 $SI \leq BX$, 转到 less_eq 执行;

(6) 若 $DI \leq DX$, 转到 below_eq 执行。

4.11 使用“SHR AX,1”将 AX 中的 D1 位移入 CF 标志, 然后用 JC/JNC 指令替代 JZ/JNZ 指令完成例 4-5 程序的功能。

4.12 在采用奇偶校验传输数据的接收端应该验证数据传输的正确性。例如, 如果采用偶校验, 那么在接收到的数据中, 其包含“1”的个数应该为 0 或偶数个, 否则说明出现传输错误。现在, 在接收端编写一个这样的程序, 如果偶校验不正确显示错误信息, 传输正确则继续。假设传送字节数据、最高位作为校验位, 接收到的数据已经保存在 Rdata 变量中。

4.13 8086 处理器的指令 CWD 将 AX 符号扩展到 DX。假若没有该指令，编程实现该指令功能。

(1) 按照符号扩展的含义编程，即：AX 最高为 0，则 DX=0；AX 最高为 1，则 DX=FFFFH。

(2) 使用移位等指令进行优化编程。

4.14 编程：首先测试字变量 VAR 的最高位，如果为 1，则显示字母“L”；如果最高位不为 1，则继续测试最低位，如果最低位为 1，则显示字母“R”；如果最低位也不为 1，则显示字母“M”。

4.15 编程：先提示输入数字“Input Number: 0~9”，然后在下一行显示输入的数字，结束；如果不是键入了 0~9 数字，就提示错误“Error!”，继续等待输入数字。

4.16 有一个首地址为 ARRAY 的 20 个字的数组，说明下列程序段的功能。

```
        mov cx,20
        xor ax,ax
        xor si,si
sumlp:   add ax,array[si]
        add si,2
        loop sumlp
        mov total,ax
```

4.17 说明如下程序段的功能。

```
        mov cx,8
        mov ah,al
next:    shr al,1
        rcr dx,1
        shr ah,1
        rcr dx,1
        loop next
        mov ax,dx
```

4.18 编程：将一个 32 位数据逻辑左移 3 位，假设该数据已经保存在 DX.AX 寄存器对中。

4.19 编程中经常要记录某个字符出现的次数。现编程记录某个字符串（假设以 0 结尾）中空格出现的次数，结果保存在 SPACE 单元中。

4.20 将一个已经按升序排列的数组（第 1 个元素最小，后面逐渐增大），改为按降序排列（即第 1 个元素最大，后面逐渐减小）。实际上只要第一个元素与最后一个元素交换，第 2 个元素与倒数第 2 个元素交换，以此类推。编程实现该功能。

4.21 编写计算 100 个 16 位正整数之和的程序。如果和不超过 16 位字的范围（65535），则保存其和到 WORDSUM，如超过，则显示“Overflow!”。

4.22 在一个已知长度的字符串中查找是否包含“BUG”子字符串。如果存在，显示“Y”，否则显示“N”。

4.23 主存中有一个 4 位压缩 BCD 码数据，保存在一个字变量中。现在需要进行显示，但要求不显示前导 0。由于位数较多，需要利用循环实现，但如何处理前导 0 和数据中间的 0 呢？不妨设置一个标记。编程实现。

4.24 已知一个字符串的长度，剔除其中所有的空格字符。请从字符串最后一个字符开始逐个向前判断、并进行处理。

4.25 第2章习题2.14在屏幕上显示ASCII码表，现仅在数据段设置表格缓冲区，编程将ASCII码值填入留出位置的表格，然后调用显示功能实现（需要利用双重循环）。

4.26 以MOVS指令为例，说明串操作指令的寻址特点，并用MOV和ADD等指令实现MOVSW的功能（假设DF=0）。

4.27 屏幕滚动程序。使用“直接写屏”方法编程，将DOS标准显示模式下的屏幕内容向上滚动一行，最后一行填充字母A。这需要将屏幕第2行（开始于 $1 \times 2 \times 80$ 的偏移地址）内容传送到第1行，第3行传送到第2行，……，最后一行（开始于 $24 \times 2 \times 80$ 的偏移地址）传送完后，填充字符A。

4.28 快速乘法程序。只使用移位和加减法指令将两个任意的16位无符号整数相乘，求出乘积（假设不超过16位）。使用这个程序显示 356×53 的结果。算法的基本思想是将较小数进行右移决定是否累加，对较大数左移位并进行累加。

4.29 素数判断程序。编写一个程序，提示用户输入一个数字，然后显示信息说明该数字是否是素数。素数（Prime）是只能被自身和1整除的自然数。

（1）采用直接简单的算法：假设输入 N ，将其逐个除以 $2 \sim N-1$ ，只要能整除（余数为0）说明不是素数，只有都不能整除才是素数。

（2）采用只对奇数整除的算法：1、2和3是素数，所有大于3的偶数不是素数，从5开始的数字只要除以从3开始的奇数，只有都不能整除才是素数。

4.30 计算素数个数程序。使用Eratosthenes筛法，求 $1 \sim 50000$ 之间共有多少个素数。

Eratosthenes筛法是希腊数学家Eratosthenes发明的算法，给出了一种找出给定范围内所有素数的快速方法。该算法要求创建一个字节数组，以如下方式将不是素数的位置标记出来：2是一个素数，从位置2开始，把所有2的倍数的位置标记1；2之后的素数3，同样将3的倍数的位置标记1；3之后下一个素数是5，同样将5的倍数的位置标记1；如此重复，直到标记所有非素数的位置。处理结束，数组中没有被标记的位置都对应一个素数。

（1）如果将该数组事先定义在数据段，则生成的可执行文件中将包括这个数组，形成很大的一个文件。MASM提供一个定义无初始化数据段的伪指令（.DATA?），在其下定义的变量不能有初值，因为它们在程序执行时才被分配存储空间。但可以利用一段循环程序将初值0全部填入其中。伪指令“.DATA?”对定义大量的无初值的变量（如数组）时特别有效。建议本程序采用这个方法。

（2）即使是在程序执行才分配存储空间，但对于求更大范围内的素数，存储空间仍然是一个编程关键。可以考虑进一步用一个二进制位表示一个数字的方法，这样1字节就可以表达8个数字，存储空间可以减少1/8。

第 5 章 模块化程序设计

当程序功能相对复杂、所有的语句序列写到一起时，程序结构将显得零乱。特别是由于汇编语言的语句功能简单，源程序显得更加冗长，降低了程序的可阅读性和可维护性。所以，编写较大型程序时，常把功能相对独立的程序段单独编写和调试，作为一个相对独立的模块供程序使用，这就是模块化程序设计。本章介绍汇编语言如何使用子程序、文件包含和宏汇编等方法来简化程序设计。

5.1 子程序结构

子程序（Subroutine）在高级语言中常被称为函数（Function）或过程（Procedure）。子程序可以实现源程序的模块化，简化源程序结构；当这个子程序被多次使用时，子程序还可以使模块得到复用，提高编程效率。本书提供了具有基本输入输出功能的多个子程序，供例题程序调用，并在本章介绍其具体应用。

5.1.1 子程序指令

程序中有些部分可以实现相同的功能，而只是参数不同；需要经常用到这些功能时，用子程序来实现就很合适。这样，不仅程序的结构更清楚，程序的维护也更方便，还有利于大程序开发时的多个程序员分工合作。

子程序通常是与主程序分开的，完成特定功能的一段程序。当主程序（调用程序 Caller）需要执行这个功能时，就可以调用该子程序（被调程序 Callee），程序转移到这个子程序的起始处执行。当运行完子程序后，再返回到调用它的主程序。子程序由主程序执行子程序调用指令 CALL 来调用；子程序执行完后，用子程序返回指令 RET 返回主程序继续执行。MASM 使用过程定义伪指令 PROC/ENDP 来编写子程序。

1. 子程序调用指令 CALL

CALL 指令用在主程序中，实现子程序的调用。子程序和主程序可以在同一个代码段内，也可以在不同的段内。因而，类似于无条件转移 JMP 指令，子程序调用 CALL 指令可以分成段内调用（近调用）和段间调用（远调用）；其目标地址又可以采用相对寻址、直接寻址或间接寻址三种方式。所以，CALL 指令分为 4 种类型，如表 5-1 所示。

表 5-1 子程序调用指令 CALL

指令类型	指令功能
段内调用、相对寻址: call label	入栈返回地址: SP=SP-2, SS: [SP]=IP 转移目标地址: IP=IP+位移量
段内转移、间接寻址: call r16/m16	入栈返回地址: SP=SP-2, SS: [SP]=IP 转移目标地址: IP=r16/m16
段间转移、直接寻址: call label	入栈返回地址: SP=SP-2, SS: [SP]=CS; SP=SP-2, SS: [SP]=IP 转移目标地址: IP=label 的偏移地址, CS=label 的段基地址
段间转移、间接寻址: call m32	入栈返回地址: SP=SP-2, SS: [SP]=CS; SP=SP-2, SS: [SP]=IP 转移目标地址: IP=m32, CS=m32+2

子程序执行结束后还要返回主程序，所以 CALL 指令不仅要同 JMP 指令一样改变 IP 和 CS 以实现转移，还要保留下一条要执行指令的地址，以便返回时重新获取。保留 IP 和 CS 的方法是压入堆栈，获取 IP 和 CS 的方法是弹出堆栈。段内调用只需入栈 16 位偏移地址，段间调用需要入栈 16 位偏移地址和 16 位段基地址。

CALL 指令实际上就是进栈 PUSH 指令和转移 JMP 指令的组合。

MASM 汇编程序根据存储模型等用户编程信息，可以自动确定是段内还是段间调用。程序员也可以采用 PTR 操作符强制改变，其方法同段内或段间转移一样。

2. 子程序返回指令 RET

子程序执行完后，应返回主程序继续执行，这一功能由 RET 指令完成。要回到主程序，只需获得离开主程序时，由 CALL 指令保存于堆栈中的指令地址即可。

实际编程应用中，RET 指令有两种书写格式：

```
ret                ;无参数返回：出栈返回地址
ret i16            ;有参数返回：出栈返回地址，SP=SP+i16
```

段内返回只需出栈 16 位偏移地址，段间返回需出栈 16 位偏移地址和 16 位段基地址，如表 5-2 所示。

表 5-2 子程序返回指令 RET

指令类型	指令功能
段内返回：ret	弹出返回地址：IP=SS: [SP], SP=SP+2
段内返回：ret i16	弹出返回地址：IP=SS: [SP], SP=SP+2 增量堆栈指针：SP=SP+i16
段间返回：ret	弹出返回地址：IP=SS: [SP], SP=SP+2; CS=SS: [SP], SP=SP+2
段间返回：ret i16	弹出返回地址：IP=SS: [SP], SP=SP+2; CS=SS: [SP], SP=SP+2 增量堆栈指针：SP=SP+i16

尽管段内返回和段间返回具有相同的汇编助记符，但汇编程序会根据子程序与主程序是否处于同一个段内，自动产生不同的指令代码；也可以分别采用 RETN 和 RETF 表示段内和段间返回。返回指令还可以带有一个立即数 i16，这时堆栈指针 SP 将增加，即 SP=SP+i16。这个特点使得程序可以方便地废除若干执行 CALL 指令以前入栈的参数。

3. 过程定义伪指令

MASM 汇编程序为配合编写子程序、中断服务程序等程序模块，设置了过程定义伪指令，由 PROC 和 ENDP 组成，基本格式如下：

```
过程名      proc
            ..... ;过程体
过程名      endp
```

其中过程名为符合语法的标识符，每个过程应该具有一个唯一的过程名。伪指令 PROC 后面还可以加上参数 NEAR 或 FAR 指定过程的调用属性：段内调用还是段间调用。在简化段定义源程序格式中，通常不需要指定过程属性，采用默认属性即可。

【例 5-1】子程序调用程序。

```
                                ;代码段，主程序
0017      B8 0001                mov ax,1
001A      BD 0005                mov bp,5
```

001D	E8 000D		call subp	
0020	B9 0003	retp1:	mov cx,3	
0023	BA 0004	retp2:	mov dx,4	
		;代码段, 子程序		
002D		subp	proc	;过程定义, 过程名为 subp
002D	55		push bp	
002E	8B EC		mov bp,sp	
0030	8B 76 02		mov si,[bp+2]	;SI=CALL 下条指令(标号 RETP1)的偏移地址
0033	BF 0023 R		mov di,offset retp2	;DI=标号 RETP2 的偏移地址
0036	BB 0002		mov bx,2	
0039	5D		pop bp	;弹出堆栈, 保持堆栈平衡
003A	C3		ret	;子程序返回
003B		subp	endp	;过程结束

为了便于理解返回地址，例程序的左边给出了列表文件的内容。

注意，用过程伪指令定义的子程序是由主程序调用才开始执行的，在源程序中应该安排在执行结束返回操作系统后（即.EXIT 语句）、END 语句前（否则不被汇编），例中没有写出返回操作系统的语句（后续例题中同样处理）。过程定义也可以安排在主程序开始执行的第 1 条语句之前。

子程序的调用和返回都要利用堆栈，如图 5-1 所示。调用时，CALL 命令先把下一条指令的偏移地址作为返回地址 IP 保存到堆栈（相当于一堆 PUSH 指令功能），然后跳转到子程序（相当于一堆 JMP 指令功能）。在本例程序中，返回地址就是“MOV CX,3”指令的地址，即标号 RETP1 地址。

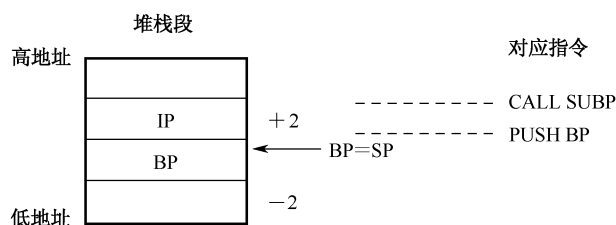


图 5-1 子程序调用的堆栈

子程序中，指令“PUSH BP”将 BP 内容保存到堆栈顶部，并使 ESP 减 2。传送指令设置 BP 等于当前堆栈指针 SP。这样，BP+2 指向堆栈保存返回地址的位置，SI 也就获得了返回地址。指令“POP BP”将当前栈顶数据传送给 BP，也就使得 BP 恢复为原来的数值（本例程序设置为 5），同时 SP 加 2。现在 SP 又指向了返回地址，执行子程序返回指令 RET，则从当前栈顶弹出这个返回地址到 IP，程序回到 CALL 的下一条指令。本程序执行结束，将显示 AX、BX、CX、DX 和 BP 依次为 1~5，SI 等于 RETP1 标号的地址，DI 等于 RETP2 标号的地址。

如果在传送 RETP2 标号地址给 DI 的指令之后，增加一条“MOV [BP+2],DI”指令。那么，由于堆栈保存返回地址的位置被设置成为 RETP2 标号地址，子程序将返回到 RETP2 标号，主程序不再执行“MOV CX,3”指令。

要验证程序执行后各个寄存器值，既可以在主程序最后调用本书提供的 16 位通用寄存器显示子程序 DISPRW，也可以载入调试程序观察。

图 5-2、图 5-3 和图 5-4 演示了子程序调用的调试过程，操作步骤如下。

```
D:\ML615>debug eg501.exe
-u
0CA9:0000 BAC0C      MOV     DX,0CAC
0CA9:0003 8EDA         MOV     DS,DX
0CA9:0005 8CD3         MOV     BX,SS
0CA9:0007 2BD0         SUB     BX,DX
0CA9:0009 D1E3         SHL     BX,1
0CA9:000B D1E3         SHL     BX,1
0CA9:000D D1E3         SHL     BX,1
0CA9:000F D1E3         SHL     BX,1
0CA9:0011 F0          CLI
0CA9:0012 8ED2         MOV     SS,DX
0CA9:0014 03E3         ADD     SP,BX
0CA9:0016 FB          STI
0CA9:0017 B80100     MOV     AX,0001
0CA9:001A BD0500     MOV     BP,0005
0CA9:001D E80A00     CALL    002A
-g id
AX=0001 BX=0010 CX=0038 DX=0CAC SP=0410 BP=0005 SI=0000 DI=0000
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=001D  NU UP EI PL NZ NA PO NC
0CA9:001D E80A00     CALL    002A
```

图 5-2 子程序调用的调试截图 1

```
0CA9:001D E80A00     CALL    002A
-t
AX=0001 BX=0010 CX=0038 DX=0CAC SP=040E BP=0005 SI=0000 DI=0000
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=002A  NU UP EI PL NZ NA PO NC
0CA9:002A 55          PUSH    BP
-t
AX=0001 BX=0010 CX=0038 DX=0CAC SP=040C BP=0005 SI=0000 DI=0000
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=002B  NU UP EI PL NZ NA PO NC
0CA9:002B 8BEC         MOV     BP,SP
-d ss:40c 40f
0CAC:0400
05 00 20 00
-t
AX=0001 BX=0010 CX=0038 DX=0CAC SP=040C BP=040C SI=0000 DI=0000
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=002D  NU UP EI PL NZ NA PO NC
0CA9:002D 8B7602     MOV     SI,[BP+02]
SS:040E=0020
-t
AX=0001 BX=0010 CX=0038 DX=0CAC SP=040C BP=040C SI=0020 DI=0000
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=0030  NU UP EI PL NZ NA PO NC
0CA9:0030 BF2300     MOV     DI,0023
```

图 5-3 子程序调用的调试截图 2

```
0CA9:0030 BF2300     MOV     DI,0023
-t
AX=0001 BX=0010 CX=0038 DX=0CAC SP=040C BP=040C SI=0020 DI=0023
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=0033  NU UP EI PL NZ NA PO NC
0CA9:0033 BB0200     MOV     BX,0002
-t
AX=0001 BX=0002 CX=0038 DX=0CAC SP=040C BP=040C SI=0020 DI=0023
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=0036  NU UP EI PL NZ NA PO NC
0CA9:0036 5D          POP     BP
-t
AX=0001 BX=0002 CX=0038 DX=0CAC SP=040E BP=0005 SI=0020 DI=0023
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=0037  NU UP EI PL NZ NA PO NC
0CA9:0037 C3          RETI
-t
AX=0001 BX=0002 CX=0038 DX=0CAC SP=0410 BP=0005 SI=0020 DI=0023
DS=0CAC ES=0C99 SS=0CAC CS=0CA9 IP=0020  NU UP EI PL NZ NA PO NC
0CA9:0020 B90300     MOV     CX,0003
```

图 5-4 子程序调用的调试截图 3

- (1) 在 DOS 环境下，输入“DEBUG EG501.EXE”带被调试文件进入调试状态。
- (2) 使用反汇编命令查看主程序代码，记住子程序调用 CALL 指令所在地址（本例是 001DH），断点执行到该位置暂停。留意此时指令指针寄存器 IP 和堆栈指针寄存器 SP，以及寄存器 AX 和 BP 的内容。
- (3) 进入子程序进行调试，需要使用跟踪命令 T，这样单步执行子程序调用 CALL 指令，进入到子程序中，等待执行子程序的第一条指令，如图 5-3 所示。此时寄存器 IP 等于子程序首地址，而 SP 减 2，堆栈压入了返回地址。
- (4) 继续单步执行“PUSH BP”指令，BP 压入堆栈保护，SP 再次减 2。查看此时堆栈内容，第一个 16 位数据是 0005H，即主程序 BP 的值；第 2 个 16 位数据是 0020H，为返回地址，即主程序 CALL 指令后的指令地址。主程序还使用了 RETP1 标号指示这个地址。
- (5) 再次单步执行指令，BP 被赋值与 SP 相同，BP+2 被指向返回地址，即 0020H。所以，继续单步执行，使得 SI 被赋值为返回地址 0020H。
- (6) 如图 5-4 所示，使用跟踪命令 T 继续单步执行，随着子程序返回指令的执行结束，指令指针 IP 又指向主程序，等于 CALL 指令后的指令地址。堆栈指针 SP 还是进入子程序前的内容，保持了堆栈平衡。BP 恢复原值，实现了寄存器的保护和恢复。

5.1.2 子程序设计

子程序也是一段程序，其编写方法与主程序一样，可以采用顺序、分支、循环结构。但是，作为相对独立和通用的一段程序，它具有一定的特殊性，需要留意几个问题。

- ① 子程序要利用过程定义伪指令声明，获得子程序名和调用属性。
- ② 子程序最后利用 RET 指令返回主程序，主程序执行 CALL 指令调用子程序。
- ③ 子程序中对堆栈的压入和弹出操作要成对使用，保持堆栈的平衡。

所谓保持堆栈平衡，是指一个程序模块压入堆栈多少字节数据，最终应该弹出多少字节，使得堆栈指针 SP 不变。

主程序 CALL 指令将返回地址压入堆栈，子程序 RET 指令将返回地址弹出堆栈。只有堆栈平衡，才能保证执行 RET 指令时当前栈顶的内容刚好是返回地址，即相应 CALL 指令压栈的内容，并返回正确的位置。

实际上，主程序中也必须保持堆栈平衡。

- ④ 子程序开始应该保护使用到的寄存器内容，子程序返回前相应地进行恢复。

因为通用寄存器数量有限，主程序和子程序可能会使用同一个寄存器。为了不影响主程序调用子程序后的指令执行，子程序应该把用到的寄存器内容保护好。常用的方法是在子程序开始，将要修改内容的寄存器顺序入栈（注意不要包括将要带回结果的寄存器）；而在子程序返回前，再将这些寄存器内容逆序弹出，恢复到原来的寄存器中。

⑤ 子程序应安排在代码段的主程序之外，最好放在主程序执行终止后的位置（返回操作系统后、汇编结束 END 伪指令前），也可以放在主程序开始执行之前的位置。

- ⑥ 子程序允许嵌套和递归。

子程序内包含有子程序的调用，这就是子程序嵌套。嵌套深度（层次）逻辑上没有限制，但受限于开设的堆栈空间。相对于没有嵌套的子程序，设计嵌套子程序并没有什么特殊要求；只是有些问题更需要注意，如正确的调用和返回、寄存器的保护与恢复等。

子程序直接或间接地嵌套调用自身，称为递归调用，含有递归调用的子程序称为递归子程序。递归子程序的设计有一定难度，但能设计出很精巧的程序。

【例 5-2】回车换行子程序。

DOS 和 Windows 平台中，实现显示器光标回到下一行首位置需要输出回车 CR（ASCII 码 0DH，光标回到当前行首位置）、换行 LF（ASCII 码 0AH，光标移到下一行、列位置不变）两个控制字符，对应 C 语言输出“\n”字符的作用。UNIX（Linux）平台只使用换行字符就可以实现光标回到下一行首位置。

本例子程序调用 2 号 DOS 字符输出功能实现回车换行。

```
dispcrlf    proc                ;回车换行子程序
             push ax            ;保护寄存器
             push dx
             mov dl,0dh          ;输出回车字符
             mov ah,2
             int 21h
             mov dl,0ah          ;输出换行字符
             mov ah,2
             int 21h
```

```

        pop dx          ;恢复寄存器
        pop ax
        ret             ;子程序返回
dispcrlf endp

```

⑦ 处理好子程序与主程序间的参数传递问题。

参数传递是子程序设计的关键和难点，5.2 节将详细讨论。

另外，为了使子程序调用更方便，编写子程序时很有必要提供适当的注释。完整的注释应该包括子程序名、子程序功能、入口参数和出口参数、调用注意事项和其他说明等。这样，程序员只要阅读了子程序的说明就可以调用该子程序，而不必关心子程序是如何编程实现该功能的。

5.2 参数传递

主程序在调用子程序时，通常需要向子程序提供一些数据，对于子程序来说就是入口参数（输入参数）；同样，子程序执行结束也要返回给主程序必要的的数据，即出口参数（输出参数、返回参数）。主程序与子程序间通过参数传递建立联系，相互配合完成任务。

传递参数的多少反映程序模块间的耦合程度。根据实际情况，子程序可以没有参数，可以只有入口参数或只有出口参数，也可以两者都有。汇编语言中，参数传递可通过寄存器、变量或堆栈来实现，参数的具体内容可以是数据本身（传递数值 By Value），也可以是数据的存储地址（传递地址 By Location）。传递数值是传递参数的一个副本，被调用程序改变这个参数时不会影响调用程序。传递地址也被称为传递引用（By Reference），使用这种方式传递参数时，被调用程序则可能修改通过地址引用的变量内容而影响程序的调用。

5.2.1 寄存器传递参数

汇编语言使用寄存器非常频繁，所以利用寄存器传递参数是最常用，也是最自然和最简单的方法，只需要将参数存于约定的寄存器即可。例如，DOS 功能调用都利用寄存器传递参数。

由于通用寄存器个数有限，这种方法对少量数据可以直接传递数值，而对大量数据只能传递地址。采用寄存器传递参数时，带有出口参数的寄存器不能进行保护和恢复，带有入口参数的寄存器则可以保护，也可以不保护，但最好能够保持一致。

注意，为了简单，一般子程序没有设计保护带入口参数的寄存器，包括 DOS 功能调用，如反映功能号的 AX，09 号调用的偏移地址 DX 等。所以，如果调用程序仍需要使用这些寄存器内容，应该在调用子程序前进行保护，使用时恢复。同时，使用低 8 位寄存器后不能保证不影响高位部分。

【例 5-3】 十六进制显示程序。

对于标准输入输出设备，操作系统往往只提供单个字符和字符串的输入输出功能。实际编程常使用十进制数据，有时也使用十六进制或者二进制数据进行输入输出，以方便用户操作。所以，底层程序设计需要实现不同数制编码间的相互转换。本章例题和习题主要围绕二进制、十进制和十六进制与 ASCII 码相互转换展开。

例如，要将数据以二进制形式输出，就需要从高位到低位依次处理，析出每个数位（数值“0”或“1”），加 30H 成为 ASCII 码（字符“0”或“1”），然后逐个字符输出（可以用 2 号 DOS 功能）或者顺序保存到主存后以字符串形式一并输出（可以用 9 号 DOS 功能），这

个转换留作习题请读者编程实现。

本例程序利用 9 号 DOS 字符串显示功能实现十六进制显示，需要以 4 个二进制位为单位，将每个十六进制数位转换为 ASCII 码。1 位十六进制数对应 4 位二进制数，具有 16 个数码：0~9、A~F。这 16 个数码依次对应的 ASCII 码是 30H~39H、41H~46H，所以十六进制数 0~9 只要加 30H 就可转换为 ASCII 码，而对 A~F（大写字母）则需要再加 7。之所以需要再加 7，是因为大写字母 A 的 ASCII 码与数字 9 的 ASCII 码相隔 7。例如，数码“B”加 30H 再加 7 等于 42H，即大写字母 B 的 ASCII 码（0BH+30H+7=42H）。

程序中需要多次进行十六进制数码转换为 ASCII 码，所以将转换过程编写成一个子程序，取名 HTOASC，使用 AL 传递入口参数（传值）和出口参数。主程序通过 AL 低 4 位将要转换的十六进制数传递给子程序，子程序转换后的 ASCII 码通过 AL 反馈给主程序。本程序的编程思想就是本书配套输入输出子程序库中显示寄存器内容 DISPRW、十六进制显示 DISPHW 等子程序所采用的方法。

```

;数据段
regw      db 'AX= ',4 dup(0),'H','$'      ;显示 AX 内容，预留 4 个字符（字节）空间
;代码段，主程序
mov ax,13ach      ;假设一个要显示的数据
xor bx,bx         ;使用 BX 相对寻址访问 REGW 字符串
mov cx,4          ;4 位十六进制数
again:  rol ax,1      ;高 4 位循环移位进入低 4 位，作为子程序的入口参数
        rol ax,1
        rol ax,1
        rol ax,1
        push ax      ;子程序利用 AL 返回结果，所以需要保存 AX 中的数据
        call htoasc   ;调用子程序
        mov regw+3[bx],al ;保存转换后的 ASCII 码
        pop ax       ;恢复保存的数据
        inc bx
        loop again
        mov dx,offset regw
        mov ah,9
        int 21h      ;显示
;代码段，子程序
htoasc    proc      ;将 AL 低 4 位表达的 1 位十六进制数转换为 ASCII 码
        and al,0fh   ;只取 AL 的低 4 位
        or al,30h    ;AL 高 4 位变成 3，实现加 30H
        cmp al,39h   ;是 0~9，还是 A~F
        jbe htoend
        add al,7      ;是 A~F，其 ASCII 码再加上 7
htoend:   ret        ;子程序返回
htoasc    endp
```

利用第 3 章学习的换码方法也可以实现 HTOASC 子程序。对应十六进制数码 0~9 和 A~F 的 ASCII 码表作为子程序只读的数据，安排在子程序代码之后。

```

;代码段，子程序
htoasc    proc
        push bx
```

```

        and ax,0fh          ;取 AL 低 4 位
        mov bx,ax
        mov al,ASCII[bx]   ;换码
        pop bx
        ret
        ;子程序的局部数据
ASCII   db '0123456789ABCDEF'
htoasc  endp

```

【例 5-4】 有符号十进制整数显示程序。

有符号整数在计算机内部以补码形式保存，要以十进制形式显示其真值，需要进行转换。

转换的算法如下：

- (1) 首先判断数据是零、正数还是负数，若为零，则显示“0”并退出。
- (2) 若为负数，显示负号“-”，求数据的绝对值。
- (3) 所得绝对值数据除以 10，余数为十进制数码，加 30H 转换为 ASCII 码保存。
- (4) 重复 (3) 步，直到商为 0 结束。
- (5) 依次从高位开始显示各位数字。

例如，对于数据 123，除以 10 后，余数 3 即为个位数，加 30H 转换为所对应的 ASCII 码。

本例程序将转换和显示编写成一个子程序，采用 AX 传递入口参数，即需要以有符号十进制形式显示的补码。子程序没有出口参数，主程序调用子程序显示若干数据。本程序的编程思想是本书的输入输出子程序库中有符号十进制显示 DISPSIW 和无符号十进制显示 DISPUIW 等子程序所采用的方法。

采用 16 位寄存器表达数据，能够显示 -32768~+32767 之间的数值。

```

        ;数据段
array   dw 6789,-1234,0,1,-9876,32767,-32768,5678,-5678,9000
count   = lengthof array
        ;代码段，主程序
        mov cx,count
        xor bx,bx
again:   mov ax,array[bx]   ;将 AX=入口参数
        call dispsiw       ;调用子程序，显示一个数据
        add bx,type array
        call dispctrlf     ;光标回车换行以便显示下一个数据
        loop again
        ;代码段，子程序
dispsiw proc               ;显示有符号十进制数的通用子程序
        push ax            ;入口参数：AX=欲显示的数据（补码）
        push bx
        push dx
        test ax,ax         ;判断数据是零、正数还是负数
        jnz dsiw1
        mov dl,'0'        ;若为零，显示“0”后退出
        mov ah,2
        int 21h
        jmp dsiw5
dsiw1:   jns dsiw2         ;若为负数，显示“-”

```

```

mov bx,ax          ;AX 数据暂存于 BX
mov dl,'-'
mov ah,2
int 21h
mov ax,bx
neg ax             ;数据求补（绝对值）
dsiw2: mov bx,10
push bx           ;10 压入堆栈，作为退出标志
dsiw3:  cmp ax,0    ;数据（商）为零，转向显示
jz dsiw4
xor dx,dx         ;扩展被除数 DX.AX
div bx            ;数据除以 10: DX.AX ÷ 10
add dl,30h        ;余数（0~9）转换为 ASCII 码
push dx           ;数据各位先低位后高位压入堆栈
jmp dsiw3
dsiw4:  pop dx      ;数据各位先高位后低位弹出堆栈
cmp dl,10         ;是结束标志 10，则退出
je dsiw5
mov ah,2          ;进行显示
int 21h
jmp dsiw4
dsiw5:  pop dx
pop bx
pop ax
ret            ;子程序返回
dispsiw endp
dispcrlf proc    ;使光标回车换行的子程序
.....         ;同例 5-2，略
dispcrlf endp

```

5.2.2 共享变量传递参数

子程序和主程序使用同一个变量名存取数据即为利用共享变量（全局变量）进行参数传递。如果变量定义和使用不在同一个程序模块中，需要利用 PUBLIC、EXTERN 进行声明（下节介绍）。如果主程序还要利用原来的变量值，则需要进行保护和恢复。

利用共享变量传递参数的子程序的通用性相对较差，但一个程序中，主程序与子程序之间或多个子程序之间通过共享变量进行参数的传递则很方便。

【例 5-5】 二进制输入程序。

二进制输入的转换原理比较简单，但需要处理输入错误的情况。利用 1 号 DOS 字符输入功能输入一个字符并判断是否合法。若为字符“0”或“1”，则合法，减去 30H 转换成数值“0”或“1”。重复转换每个字符的同时，需要将前一次的数值左移 1 位，并与新数值进行组合。如果输入了非“0”或“1”的字符，或者超过了数据位数，则提示错误重新输入。

本例程序将二进制输入编写成一个子程序，输入的二进制数据使用共享变量返回（即出口参数）。子程序没有入口参数，主程序调用子程序输入若干数据。本程序的编程思想是本书子程序库中二进制输入 READBW 等子程序所采用的方法。

```

;数据段
count      = 5
array      dw count dup(0)
temp       dw ?           ;共享变量

;代码段，主程序
mov cx,count
mov bx,offset array

again:     call readbw      ;调用子程序，输入一个数据
           mov ax,temp      ;获得出口参数
           mov [bx],ax      ;存放到数据缓冲区
           add bx,type array
           call dispctrlf   ;光标回车换行以便输入下一个数据
           loop again

;代码段，子程序
readbw     proc            ;二进制输入子程序
           push ax          ;出口参数：共享变量 TEMP
           push bx
           push cx

rdbw1:     xor bx,bx        ;BX 用于存放二进制结果
           mov cx,16        ;限制输入字符的个数

rdbw2:     mov ah,1         ;输入一个字符
           int 21h
           cmp al,'0'       ;检测键入字符是否合法
           jb rderr         ;不合法则返回重新输入
           cmp al,'1'
           ja rderr
           sub al,'0'       ;对输入的字符进行转化
           shl bx,1         ;BX 的值乘以 2
           or bl,al         ;BL 和 AL 相加
           loop rdbw2       ;循环键入字符
           mov temp,bx      ;把 BX 的二进制结果存放 TEMP 返回
           pop cx
           pop bx
           pop ax
           ret

rderr:     push ds          ;保护 DS
           mov ax,cs        ;因信息保存在代码段，所以需要设置 DS=CS
           mov ds,ax
           lea dx,errmsg    ;显示错误信息
           mov ah,9
           int 21h
           pop ds           ;恢复 DS
           jmp rdbw1

errmsg     db 0dh,0ah,'Input error, enter again: $'
readbw     endp
dispctrlf  proc            ;使光标回车换行的子程序
           .....          ;同例 5-2，略
dispctrlf  endp

```

【例 5-6】 有符号十进制数输入程序。

我们习惯使用十进制输入数据，但计算机内部采用二进制编码表达和处理，所以需要进行转换。编写一个子程序实现从键盘输入一个有符号十进制数。输入时，负数用“-”引导，正数直接输入或用“+”引导；正确的数码是 0~9，其他字符默认为输入数据结束。数据输入过程中，需要将原输入数码乘以 10，然后与新输入数据相加，其中还包含将 ASCII 码转换为二进制数的过程，其算法（对应的汇编语言程序流程图见图 5-5）如下。

（1）首先判断输入的是正数还是负数，并用一个寄存器进行记录。

（2）输入 0~9 数字（ASCII 码），并减去 30H 转换为二进制数；因为刚输入的数码作为十进制个位，则前面输入的数值依次向十位、百位等移动一位，即乘以 10。

（3）然后将前面输入的数值乘以 10，并与刚输入的数字相加得到新的数值。

（4）重复（2）、（3）步，直到输入一个非数字字符结束。

（5）如果是负数则进行求补，转换成补码；否则直接将数值保存。

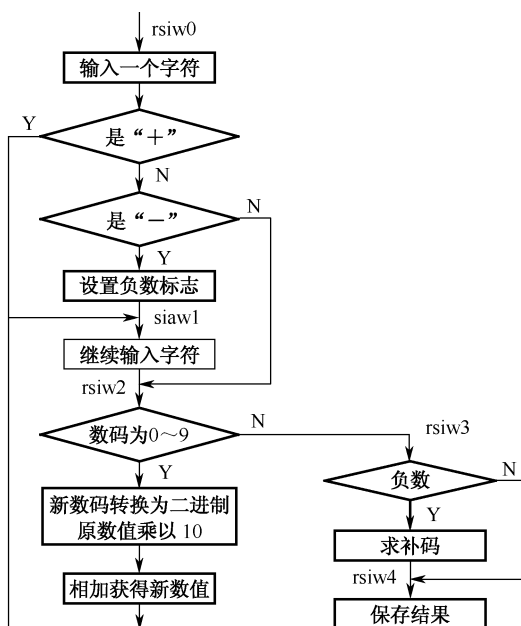


图 5-5 有符号十进制输入流程图

本例采用 64 位寄存器表达结果数值，所以输入的数据范围是-32768~+32767。为了简化问题，子程序中没有处理输入数据超出范围的问题。

该子程序不需要入口参数，出口参数是补码表示的 16 位有符号二进制数据，通过共享变量 TEMP 传递。主程序调用该子程序输入若干数据。本程序的编程思想是 I/O 子程序库中有符号十进制输入 READSIW 和无符号十进制输入 READUIW 等子程序所采用的方法。

```

;数据段
count      = 5
array      dw count dup(0)
temp       dw ?           ;共享变量

;代码段，主程序
mov cx,count
mov bx,offset array
  
```


again:	call readsiw	;调用子程序，输入一个数据
	mov ax,temp	;获得出口参数
	mov [bx],ax	;保存到数据缓冲区
	add bx,2	
	call dispCrLf	;分行
	loop again	
	;代码段，子程序	
readsiw	proc	;输入有符号十进制数的通用子程序
	push ax	;出口参数：变量 TEMP=补码表示的二进制数值
	push bx	;说明：负数用“-”引导
	push cx	
	xor bx,bx	;BX 保存结果
	xor cx,cx	;CX 为正负标志，0 为正，-1 为负
rsiw0:	mov ah,1	;输入一个字符
	int 21h	
	cmp al,'+'	;是“+”，继续输入字符
	jz rsiw1	
	cmp al,'-'	;是“-”，设置-1 标志
	jnz rsiw2	
	mov cx,-1	
rsiw1:	mov ah,1	;继续输入字符
	int 21h	
rsiw2:	cmp al,'0'	;不是 0~9 之间的字符，则输入数据结束
	jb rsiw3	
	cmp al,'9'	
	ja rsiw3	
	sub al,30h	;是 0~9 之间的字符，则转换为二进制数
	xor ah,ah	;AL 零位扩展为 AX
	shl bx,1	;利用移位和加法实现数值乘 10: $BX \leftarrow BX \times 10$
	mov dx,bx	;参见例 3-8
	shl bx,1	
	shl bx,1	
	add bx,dx	
	add bx,ax	;已输入数值乘 10 后，与新输入数值相加
	jmp rsiw1	;继续输入字符
rsiw3:	cmp cx,0	;是负数，进行求补
	jz rsiw4	
	neg bx	
rsiw4:	mov temp,bx	;设置出口参数
	pop cx	
	pop bx	
	pop ax	
	ret	;子程序返回
readsiw	endp	
dispCrLf	proc	;使光标回车换行的子程序
	;同例 5-2，略
dispCrLf	endp	

5.2.3 堆栈传递参数

参数还可以通过堆栈这个临时存储区进行传递。主程序先将入口参数压入堆栈，子程序再从堆栈中取出参数；出口参数通常不使用堆栈传递。高级语言进行函数调用时提供的参数，实质上也是利用堆栈传递的；高级语言还利用堆栈创建局部变量。保存参数和局部变量的堆栈区域被称为堆栈帧（Stack Frame），函数调用时建立，返回后消失。

【例 5-7】 计算有符号数的平均值程序。

假设有一个 16 位有符号整型数组，主程序调用子程序求其平均值，并显示结果。子程序需要两个参数：数组指针和元素个数，通过堆栈进行传递。子程序通过寄存器 AX 返回平均值。

```

;数据段
array      dw 675, 354, -34, 198, 267, 0, 9, 2371, -67, 4257
;代码段，主程序
mov ax,lengthof array
push ax    ;压入数据个数
mov bx,offset array
push bx    ;压入数组的偏移地址
call mean  ;调用求平均值子程序，出口参数：AX=平均值（整数部分）
add sp,4   ;平衡堆栈（压入了 4 字节数据）
call dispsiw ;显示
;代码段，子程序
mean       proc ;计算 16 位有符号数平均值子程序
push bp    ;入口参数：顺序压入数据个数和数组偏移地址
mov bp,sp  ;出口参数：AX=平均值
push bx    ;保护寄存器
push cx
push dx
mov bx,[bp+4] ;BX=堆栈中取出的偏移地址
mov cx,[bp+6] ;CX=堆栈中取出的数据个数
xor ax,ax   ;AX 保存和值
mean1:     add ax,[bx] ;求和
           add bx,type array ;指向下一个数据
           loop mean1 ;循环
           cwd ;将累加和 AX 符号扩展到 DX
           idiv word ptr [bp+6] ;有符号数除法，AX=平均值（余数在 DX 中）
           pop dx ;恢复寄存器
           pop cx
           pop bx
           pop bp
           ret
mean       endp
dispsiw    proc ;有符号十进制显示子程序：AX=欲显示的数据
           ..... ;同例 5-4，略
dispsiw    endp
```

上述程序执行过程中利用堆栈传递参数的情况如图 5-6 所示。主程序依次压入数据个数 (LENGTHOF ARRAY) 和数组偏移地址 (OFFSET ARRAY)，子程序调用时压入返回地址 (IP)。进入子程序后，压入 BP 寄存器保护；然后设置基址指针 BP 等于当前堆栈指针 SP，这样利用 BP 相对寻址（默认指向堆栈段）可以存取堆栈段中的数据。主程序压入了两个参数，使用了堆栈区的 8 字节的存储空间；为了保持堆栈的平衡，主程序在调用 CALL 指令后用一条“ADD SP,4”指令平衡堆栈。这就是调用程序平衡堆栈。平衡堆栈也可以规定由被调用程序实现，返回指令则采用“RET 4”，使 SP 加 4。

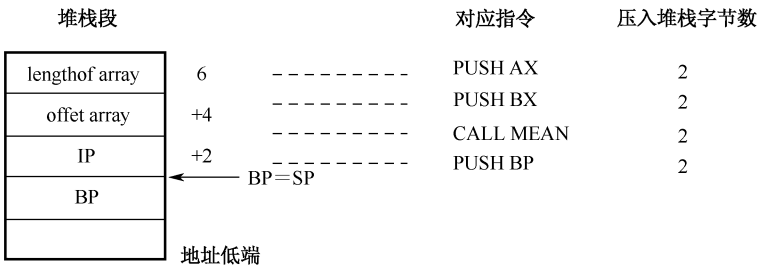


图 5-6 利用堆栈传递参数

由此可见，由于堆栈采用“先进后出”的原则进行存取，且返回地址和保护寄存器等也要存于堆栈，因此，用堆栈传递参数时要时刻注意堆栈的分配情况，保证参数的正确存取以及子程序的正确返回。为了降低利用堆栈传递参数的编程难度，MASM 从 6.0 开始将过程定义伪指令 PROC 进行了扩展，并引入了过程声明 PROTO 伪指令和过程调用 INVOKE 伪指令。利用这些高级语言的特性，程序员就可以不必关心具体的堆栈位移，直接使用变量名（详见 6.3 节）。

为了简化问题，上述子程序没有处理求和过程中可能的溢出，这是一个潜在的错误。为了避免有符号数据运算的溢出，将被加数进行符号扩展，得到倍长数据（大小没有变化），然后求和。我们使用二进制 16 位表示数据个数，最大是 2^{16} ，这样扩展到 32 位二进制数表达累加和，不再会出现溢出（考虑极端情况：数据全是一 2^{15} ，共有 2^{16} 个，求和结果是一 2^{31} ，32 位数据仍然可以表达）。改进的子程序如下：

```

;代码段，子程序
mean proc ;计算 16 位有符号数平均值子程序
push bp ;入口参数：顺序压入数据个数和数据缓冲区偏移地址
mov bp,sp ;出口参数：AX=平均值
push bx ;保护寄存器
push cx
push dx
push si
push di
mov bx,[bp+4] ;BX=堆栈中取出的偏移地址
mov cx,[bp+6] ;CX=堆栈中取出的数据个数
xor si,si ;SI=求和的低 16 位值
mov di,si ;DI=求和的高 16 位值
mean1: mov ax,[bx] ;AX=取出一个数据
cwd ;AX 符号扩展到 DX
add si,ax ;求和低 16 位

```

```

        adc di,dx                ;求和高 16 位
        add bx,2                ;指向下一个数据
        loop mean1              ;循环
        mov ax,si               ;累加和在 DX.AX
        mov dx,di
        idiv word ptr [bp+6]    ;有符号数除法, AX=平均值 (余数在 DX 中)
        pop di                  ;恢复寄存器
        pop si
        pop dx
        pop cx
        pop bx
        pop bp
        ret
mean     endp

```

上述程序还隐含一个问题, 如果将 0 作为元素个数压入堆栈, 除法指令将产生除法错误异常。改进的方法可以是在个数为 0 时, 直接赋值 0 作为返回结果。

进一步, 还可以将计算平均值程序载入调试程序, 观察堆栈传递参数的动态过程。图 5-7 和图 5-8 演示堆栈传递参数程序的调试过程, 操作步骤如下。

```

-g 17
AX=0000 BX=0020 CX=009E DX=0CB1 SP=0420 BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=0017  NO UP EI PL NZ NA PO NC
0CA9:0017 B80A00  MOV     AX,000A
-g 1f
AX=000A BX=000A CX=009E DX=0CB1 SP=041C BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=001F  NO UP EI PL NZ NA PO NC
0CA9:001F E80A00  CALL    002C
-t 2
AX=000A BX=000A CX=009E DX=0CB1 SP=041A BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=002C  NO UP EI PL NZ NA PO NC
0CA9:002C 55      PUSH   BP
AX=000A BX=000A CX=009E DX=0CB1 SP=0418 BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=002D  NO UP EI PL NZ NA PO NC
0CA9:002D 8BEC     MOV     BP,SP
-d ss:418 41f
0CB1:0418                00 00 22 00 0A 00 0A 00  ..".....

```

图 5-7 堆栈传递参数的调试截图 1

```

-g 49
AX=0323 BX=000A CX=009E DX=0CB1 SP=041A BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=0049  NO UP EI PL NZ NA PE NC
0CA9:0049 C3      RET
-t
AX=0323 BX=000A CX=009E DX=0CB1 SP=041C BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=0022  NO UP EI PL NZ NA PE NC
0CA9:0022 83C404  ADD     SP,+04
-t
AX=0323 BX=000A CX=009E DX=0CB1 SP=0420 BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=0025  NO UP EI PL NZ AC PO NC
0CA9:0025 E82200  CALL    004A
-p
003
AX=0323 BX=000A CX=009E DX=0CB1 SP=0420 BP=0000 SI=0000 DI=0000
DS=0CB1 ES=0C99 SS=0CB1 CS=0CA9 IP=0028  NO UP EI PL ZR NA PE NC
0CA9:0028 B44C     MOV     AH,4C

```

图 5-8 堆栈传递参数的调试截图 2

(1) 将生成的可执行文件载入 DEBUG 调试程序, 使用反汇编命令查看主程序, 并注意有关指令 (例如 CALL) 的地址, 以便进行断点执行。

(2) 命令 “G 17” 断点执行实现 “.STARTUP” 语句功能, 设置数据段 DS、堆栈段 SS 以及堆栈指针 SP, 本例 SP=0420H。

(3) 命令 “G 1F” 断点执行到调用子程序, 实现两个参数压入堆栈, SP 减 4, 等于 041CH。

(4) 单步执行两条指令, 进入子程序, 保护 BP, 堆栈又被压入返回地址和 BP 内容, SP 再次减 4, 等于 0418H。可以利用显示命令 D 观察堆栈中的数据, 其中返回地址为 0022H, 堆栈指针加 4 指向最后压入堆栈的参数, 加 6 指向本例中首先压入的参数, 即数组元素个数。

(5) 进入子程序后, 使用反汇编命令则可以查看子程序代码。接着, 可以单步执行仔细观察计算平均值的过程, 以及通过 BP 读取堆栈中的传递参数, 也可以断点执行到返回主程序前, 如图 5-8 所示。

(6) 继续单步执行, 观察子程序返回指令的作用。此时, 指令指针 IP 变成 CALL 指令后的指令地址, 即返回地址(本例是 0022H)。SP 也恢复调用子程序之前的值(本例是 041CH), 但还不是主程序初始值(本例是 0420H), 所以需要执行“ADD SP, 4”才能实现堆栈平衡。

(7) 有兴趣的话, 还可以继续使用跟踪命令 T 进入显示子程序。如果使用继续命令 P, 则可以在主程序层次实现单步执行, 完成子程序调用, 显示求得平均值(本例是 803)。

5.3 多模块程序结构

对经常用到的应用问题, 可以编写一个通用的子程序在需要时调用; 看似无法入手的大型处理过程可以逐步分解, 划分成一个个能够解决的模块。子程序实现了程序的模块化, 使得程序结构简洁清晰。还可以进一步进行单独编辑并汇编子程序, 生成目的代码文件或子程序库, 使用多个模块组成完整的程序。

5.3.1 源文件包含

为了方便编辑大型源程序, 可以将整个源程序合理地分放在若干个文本文件中。例如, 可以将各种常量定义、声明语句等组织在包含文件中(一般用扩展名 INC, 类似 C/C++ 语言的头文件); 也可以把一些常用的或有价值的宏定义存放在宏定义文件中(一般用扩展名 MAC, 详见下节); 还可以将常用的子程序编辑成汇编语言源文件。

这样, 主体源程序文件只要使用源文件包含伪指令 INCLUDE, 就能将它们结合成一体, 按照一个源程序文件进行汇编连接, 形成可执行文件。其格式为:

INCLUDE 文件名

文件名应符合操作系统规范, 必要时还应含有路径, 用于指明文件的存储位置; 如果没有路径名, 汇编程序将在默认目录、当前目录和指定目录下寻找。

源文件包含的一定是文本文件, 汇编程序在对 INCLUDE 伪指令进行汇编时将它指定的文本文件内容插入该伪指令所在位置, 与其他部分同时进行汇编。但是需要明确的是, 利用 INCLUDE 伪指令包含其他文件, 其实质仍然是一个源程序, 只不过是分在了几个文件书写; 被包含的文件不能独立汇编, 是依附主程序而存在的。所以, 合并的源程序之间的各种标识符, 如标号和名字等, 应该统一规定, 不能发生冲突。

【例 5-8】 存储器数据显示程序。

为了方便查看主存内容, 实现一个存储器数据显示程序。用户输入一个 16 位存储器地址(用 4 位十六进制表示), 程序分别用十六进制和十进制显示该字单元的数据。

为了说明源文件包含方法, 将数据段内容书写在一个文件中, 涉及的十六进制输入和输出、十进制输出编写成 3 个子程序编辑在一个文件中, 主程序文件包含它们实现指定存储单元内容的显示。如下给出了各文件内容, 程序完整, 不需套用框架模板文件。

;文件名: eg508.inc, 例 5-8 程序的数据段内容

```
                .data                ;数据段
var             dw 24bdh
```

```

inmsg      db 'Enter Memory Address: $'
outmsg1    db 'Memory Data In HexDecimal: $'
outmsg2    db 'Memory Data In Signed Decimal: $'
temp       dw ?                ;共享变量

```

;文件名: eg508.asm, 例 5-8 主程序

```

.model small
.stack
include eg508.inc    ;源文件包含: 数据段文件
.code               ;代码段, 主程序
.startup
mov temp,offset var
call disphw         ;十六进制输出, 显示变量 VAR 地址以便输入
call dispctrlf      ;分行

mov dx,offset inmsg
mov ah,9
int 21h
call readhw         ;输入存储器地址, 结果返回 AX
call dispctrlf
mov si,ax
mov bx,[si]         ;BX=存储器数据
mov dx,offset outmsg1
mov ah,9
int 21h
mov temp,bx         ;共享变量传递参数
call disphw         ;十六进制输出
call dispctrlf
mov dx,offset outmsg2
mov ah,9
int 21h
mov ax,bx           ;寄存器传递参数
call dispziw        ;十进制输出
.exit               ;主程序结束, 退出
include eg508s.asm  ;源文件包含: 子程序文件
end

```

;文件名: eg508s.asm, 例 5-8 程序的子程序

```

readhw      proc                ;十六进制输入子程序
push bx     ;出口参数: BX=输入的数据
push cx

rdhw1:      xor bx,bx           ;BX 用于存放十六进制结果
mov cx,4    ;限制输入字符的个数

rdhw2:      mov ah,1
int 21h     ;输入一个字符
cmp al,'0'  ;检测键入字符是否合法
jnb rderr   ;不合法, 则返回重新输入

```

	cmp al,'9'	
	jbe rdhw4	;输入数码: 0~9, 减 30H
	cmp al,'A'	
	jb rderr	
	cmp al,'F'	
	jbe rdhw3	;输入大写字母: A~F, 减 7 后、再减 30H
	cmp al,'a'	
	jb rderr	
	cmp al,'f'	;输入小写字母: a~f, 减 20H、减 7 后、再减 30H
	ja rderr	
	sub al,20h	;减 20H
rdhw3:	sub al,7	;减 7
rdhw4:	sub al,30h	;减 30H
	shl bx,1	;BX 左移 4 位, 对应十六进制 1 位
	shl bx,1	
	shl bx,1	
	shl bx,1	
	or bl,al	;BL 和 AL 相加
	loop rdhw2	;循环键入字符
	mov ax,bx	;通过 AX 返回结果
	pop cx	
	pop bx	
	ret	
rderr:	push ds	;保护 DS
	mov ax,cs	;因信息保存在代码段, 所以需要设置 DS=CS
	mov ds,ax	
	lea dx,errmsg	;显示错误信息
	mov ah,9	
	int 21h	
	pop ds	;恢复 DS
	jmp rdhw1	
errmsg	db 0dh,0ah,'Input error, enter again: \$'	
readhw	endp	
disphw	proc	;十六进制输出子程序 (参考例 5-3)
	push ax	;入口参数: TEMP=输出的数据
	push cx	
	push dx	
	mov dx,temp	;使用共享变量传递的参数
	mov cx,4	;4 位十六进制数
dphw1:	rol dx,1	;高 4 位循环移位, 进入低 4 位
	rol dx,1	
	rol dx,1	
	rol dx,1	
	push dx	;保护 DX 中的数据
	and dl,0fh	;只取 AL 的低 4 位
	or dl,30h	
	cmp dl,39h	

```

                                jbe dphw2
                                add dl,7
dphw2:                          mov ah,2
                                int 21h           ;显示一个数位
                                pop dx             ;恢复保存的数据
                                loop dphw1
                                pop dx
                                pop cx
                                pop ax
                                ret
disphw                          endp
dispsiw                         proc             ;有符号十进制显示子程序: AX=欲显示的数据
                                .....           ;同例 5-4, 略
dispsiw                         endp
dispcrlf                       proc             ;使光标回车换行的子程序
                                .....           ;同例 5-2, 略
dispcrlf                       endp

```

十六进制输入的原理类似二进制输入,只是合法的字符有数码 0~9(减 30H 转为数值)、大写字母 A~F(需要再减 7,例如大写字母 A 的 ASCII 码为 41H,41H-7-30H=0AH=10,表达数值 10)和小写字母 a~f(先减 20H 转换为大写字母,然后转换为数值 10~15)。另外,十六进制 1 位对应二进制 4 位,所以输入 1 位十六进制数后需要左移二进制 4 位存放新的数据。

完成这 3 个文件的编辑后,把它们保存在同一个目录下,然后只针对主程序进行汇编连接即可。在生成的列表文件中被包含的源程序行增加了一个字母“C”标示。

由于 DOS 支持的地址空间只有 1MB,所以本例程序运行时输入的地址必须小于等于 FFFCH,否则将出现错误,在 Windows 操作系统的 DOS 模拟窗口将被关闭。所以,主程序开始安排了显示变量 VAR 所在地址的片段。这样,执行本程序时,可以按照这个地址进行输入,随后显示“24BD”,正是本程序安排的变量值,其十进制真值是“9405”,如图 5-9 所示。



```

D:\ML615>eg508
000A
Enter Memory Address: 000a
Memory Data In HexDecimal: 24BD
Memory Data In Signed Decimal: 9405
D:\ML615>_

```

图 5-9 例 5-8 程序运行情况

5.3.2 模块连接

为了使子程序更加通用和便于复用,可以将子程序单独编写成一个源程序文件,经过汇编之后形成目标模块 OBJ 文件,即为子程序模块。这样,某个程序使用到该子程序,只要在连接时输入子程序模块文件名就可以了。

将子程序汇编成独立的模块,编写源程序文件时,需要注意几个问题。

① 子程序文件中的子程序名、定义的共享变量名要用共用伪指令 PUBLIC 声明后才可为其他程序使用。子程序使用的其他模块或主程序中定义的子程序或共享变量,也要用外部伪指令 EXTERN 进行声明。主程序文件同样需要进行声明,即本程序定义的共享变量、过程等需要用 PUBLIC 声明为共用;使用其他程序定义的共享变量、过程等需要用 EXTERN 声

明为来自外部。

PUBLIC 标识符 [标识符 ...] ; 定义标识符的模块使用

EXTERN 标识符:类型 [标识符:类型 ...] ; 调用标识符的模块使用

其中, 标识符是变量名、过程名等; 类型是 NEAR、FAR (过程) 或 BYTE、WORD、DWORD (变量) 等。在一个源程序中, **PUBLIC** 和 **EXTERN** 语句可以有 multiple 条。

MASM 默认过程名为共用, 而变量和符号私有。

② 子程序必须在代码段中, 与主程序文件采用相同的存储模型, 但没有主程序那样的开始执行点和结束执行点。还需要特别处理好子程序与主程序之间的参数传递问题, 可以采用寄存器、共享变量或堆栈等方法进行传递。利用共享变量传递参数, 要利用 **PUBLIC** 和 **EXTERN** 进行声明。

针对例 5-8 的存储器数据显示程序, 现在使用子程序模块连接方法进行开发。

将十六进制输入和输出、十进制输出等 4 个子程序单独编辑成一个文件 (如果每个子程序都很大, 也可以分成 4 个文件), 只要把上节的子程序文件 EG508S.ASM 简单修改如下:

; 文件名: eg508es.asm, 用于例 5-8 程序进行模块连接子程序

```
.model small
public readhw,disphw,dispsiw,dispcrlf    ;子程序共用
extern temp:word                        ;外部变量
.code                                    ;代码段, 子程序
...                                     ;同 EG508S.ASM 文件, 略
end                                       ;汇编结束
```

利用 **PUBLIC** 语句声明 4 个子程序可以为其他程序共用。利用 **EXTERN** 语句说明变量 TEMP 来自外部其他文件, 本程序可以使用, 其类型是字 WORD。

主程序文件需要删除包含子程序文件的语句时, 可增加如下语句:

```
extern readhw:near,disphw:near,dispsiw:near,dispcrlf:near    ;外部子程序
public temp                                                  ;变量共用
```

现在需要将主程序文件 (EG508E.ASM) 和子程序文件 (EG508ES.ASM) 分别汇编, 形成模块文件 (不要忘记参数 “/c”):

```
ml /c eg508e.asm
ml /c eg508es.asm
```

然后用连接程序 LINK 将两个 OBJ 文件连接在一起 (用 “+” 或者空格分隔多个模块文件):

```
link eg508e.obj+eg508es.obj;
```

如果希望进行源程序文件调试, 可以再加上生成调试信息的参数 (详见第 1 章)。

5.3.3 子程序库

当子程序模块很多时, 可以把它们统一管理起来, 存入一个或多个子程序库中。子程序库文件 (.LIB) 就是子程序模块的集合, 其中存放着各子程序的名称、目标代码以及有关定位信息等。

编写存入库的子程序与子程序模块中的要求一样, 只是为方便调用, 更加严格, 最好遵循一致的规则。例如, 参数传递方法、子程序调用类型、存储模型、寄存器保护措施和堆栈平衡措施等都最好相同。子程序文件编写完成, 汇编形成目标模块; 然后利用库管理工具程序 LIB.EXE, 把子程序模块逐个加入到库中, 连接时就可以使用了。

例如，将例 5-8 的子程序文件汇编成模块文件，使用如下命令创建子程序库文件：

```
lib eg508e.lib eg508es.obj
```

其中，前一个文件的扩展名为 LIB，是保存子程序及其有关连接信息的子程序库；后一个文件是将要进入子程序库的模块文件，多个文件可以使用“+”连接。如果用“-”，则是将该模块从子程序库删除。

使用子程序库，在连接主程序模块时需要提供子程序库文件名（本例是 EG508E.LIB）：

```
link eg508e.obj+eg508e.lib;
```

或者在提示输入子程序库（Libraries[.lib]:）时输入子程序库文件名即可。

有了子程序库，还可以直接在主程序源文件中用库文件包含伪指令 INCLUDELIB 进行说明，这样就不用在连接时输入库文件名，操作起来更方便，其格式为：

```
includelib 文件名
```

这里的文件必须是库文件，文件名要符合操作系统规范，类同源文件包含伪指令。

组合两种文件包含，以及宏汇编等方法，可以精简程序框架，简化程序设计。例如，本书为方便使用子程序提供的 IO.INC 中，涉及子程序库的语句如下：

```
;declare procedures for inputting and outputting charactor or string
extern readc:near,readmsg:near
extern dispcc:near,dispmsg:near,dispctrlf:near
...
;declare I/O libraries
includelib io.lib
```

文件包含封装了复杂的、深入的内容，利于大家从易到难展开汇编语言的学习。

5.4 宏结构

宏汇编、重复汇编和条件汇编都用于简化源程序结构，本书将其统称为宏结构。

5.4.1 宏汇编

宏（Macro）是具有宏名的一段汇编语句序列。宏需要先定义，然后在程序中进行宏调用。由于调用形式类似其他指令，所以常称其为宏指令。宏指令实际上是一段语句序列的缩写，汇编程序将用对应的语句序列替代宏指令，即展开宏指令。因为宏指令是在汇编过程中实现的宏展开，所以常称为宏汇编。

1. 宏定义和宏调用

宏定义由一对宏汇编伪指令 MACRO 和 ENDM 来完成，其格式如下：

```
宏名      macro [形参表]
          .....          ;;宏定义体
          endm
```

其中，宏名是符合语法的标识符，同一源程序中该名字定义唯一。宏定义体中不仅可以是硬指令组成的执行性语句序列，还可以是伪指令组成的指示性语句序列。可选的形参表给出了宏定义中用到的形式参数，每个形式参数之间用逗号分隔。

例如，将调用字符串显示子程序 DISPMSG 编写成一个宏 WRITESTRING，其中宏的参数是定义字符串的名称 MSG：

```

WriteString macro msg
    push ax
    push dx
    lea dx,msg
    mov ah,9
    int 21h
    pop dx
    pop ax
endm

```

宏定义之后就可以使用它，即宏调用。在使用宏指令的位置写下宏名，后跟实体参数；如果有多个参数，应按形参顺序填入实参，也用逗号分隔。格式如下：

宏名 [实参表]

例如，使用上面宏定义的宏调用指令是：

```
WriteString msg ;MSG 是程序中定义的字符串名称
```

在汇编时，宏指令被汇编程序用宏定义的代码序列替代。例如，上面宏指令被展开为：

```

push ax
push dx
lea dx,msg
mov ah,9
int 21h
pop dx
pop ax

```

宏展开的具体过程是：当汇编程序扫描源程序遇到已有定义的宏调用时，即用相应的宏定义体取代源程序的宏指令，同时用位置匹配的实参对形参进行取代。实参与形参的个数可以不等，多余的实参不予考虑，缺少的实参对相应的形参做“空”处理；另外，汇编程序不对实参和形参进行类型检查，取代时完全是字符串的替代，至于宏展开后是否有效，则由汇编程序翻译时进行语法检查。由此可见，宏调用不需要控制转移与返回，而是将相应的程序段复制到宏指令的位置、嵌入源程序。

宏定义允许嵌套，即宏定义体内可以有宏定义，对这样的宏进行调用时需要多次分层展开。宏定义内也允许递归调用，这种情况需要用到后面将介绍的条件汇编指令给出递归出口条件。

宏需要先定义后使用，不必在任何段中，所以宏定义通常书写在源程序的开头。为了使宏定义为多个源程序使用，可以将常用的宏定义单独写成一个宏定义文件。要使用这些宏时，只要采用源文件包含伪指令 **INCLUDE** 将它们结合成一体。如果程序中并没有使用定义的宏，则汇编后的程序中也不会包含宏定义的语句。

如果宏定义中使用了寄存器，最好也像子程序一样进行保护和恢复，以便更通用。

【例 5-9】 状态标志显示程序。

利用处理器指令 **PUSHF** 可以先将当前 16 位标志寄存器内容压入堆栈，再从堆栈弹出就能够逐个数位进行显示。本程序仅显示我们经常关注的 6 个状态标志，如图 5-10 所示，注意从高位开始 **OF**、**SF**、**ZF**、**AF**、**PF** 和 **CF** 标志依次是数据的 D11、D7、D6、D4、D2 和 D0 位（详见 1.2 节的图 1-3）。只要将欲显示的标志位移入 **CF**，加 30H 转换为 ASCII 码即可显示。

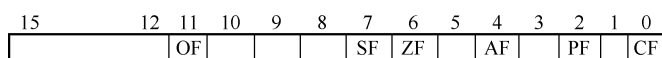


图 5-10 状态标志在标志寄存器的位置

由于每个位的处理类似，可以考虑编写成宏，不同的位数用做参数代入。

```

;宏定义
rfbit macro bit1,bit2
    xor bx,bx          ;BX 清 0，用于保存字符
    mov cl,bit1        ;将某个标志左移 BIT1 位
    rol ax,cl          ;进入当前 CF
    adc bx,30h         ;转换为 ASCII 字符
    mov rfmsg+bit2,bl   ;保存于字符串 BIT2 位置
endm

;数据段
rfmsg db 'OF=0, SF=0, ZF=0, AF=0, PF=0, CF=0',13,10,'$'

;代码段
mov ax,50
sub ax,80              ;假设一个运算
pushf                  ;将标志位寄存器的内容（保护上条指令影响的状态标志）压入堆栈
pop ax                 ;将标志位寄存器的内容存入 AX

rfbit 5,3               ;显示 OF（原来的 OF 需左移 5 位，进入当前 CF）
rfbit 4,9               ;显示 SF（原来的 SF 再左移 4 位，进入当前 CF）
rfbit 1,15              ;显示 ZF（原来的 ZF 再左移 1 位，进入当前 CF）
rfbit 2,21              ;显示 AF（原来的 AF 再左移 2 位，进入当前 CF）
rfbit 2,27              ;显示 PF（原来的 PF 再左移 2 位，进入当前 CF）
rfbit 2,33              ;显示 CF（原来的 CF 再左移 2 位，进入当前 CF）
mov dx,offset rfmsg
mov ah,9
int 21h

```

打开汇编时生成的列表文件，可以看到前两个宏调用展开为：

```

rfbit 5,3               ;显示 OF（原来的 OF 需左移 5 位，进入当前 CF）
1    xor bx,bx          ;BX 清 0，用于保存字符
1    mov cl,5           ;将某个标志左移 BIT1 位
1    rol ax,cl          ;进入当前 CF
1    adc bx,30h         ;转换为 ASCII 字符
1    mov rfmsg+3,bl     ;保存于字符串 BIT2 位置
rfbit 4,9               ;显示 SF（原来的 SF 再左移 4 位，进入当前 CF）
1    xor bx,bx          ;EBX 清 0，用于保存字符
1    mov cl,4           ;将某个标志左移 BIT1 位
1    rol ax,cl          ;进入当前 CF
1    adc bx,30h         ;转换为 ASCII 字符
1    mov rfmsg+9,bl     ;保存于字符串 BIT2 位置

```

其中，带数字“1”的语句为相应的宏定义体（数字代表宏展开的层数）。

2. 宏的参数和宏的操作符

宏的参数功能强大，既可以无参数，又可以带有一个或多个参数；而且参数的形式非常灵活，可以是常数、变量、存储单元、指令（操作码）或它们的一部分，也可以是表达式；宏的参数可以设置为不可缺少（使用“: REG”说明），还可以预设默认值。

运用宏时还经常需要一些宏操作符配合，如表 5-3 所示。

表 5-3 宏操作符

宏操作符	作用或含义
&	替换操作符，用于将参数与其他字符分开。如果参数紧接在其他字符之前或之后，或者参数出现在带引号的字符串中，就必须使用该伪操作符
<>	字符串传递操作符，用于括起字符串。在宏调用中，如果传递的字符串实参数含有逗号、空格等间隔符号，则必须用这对操作符，以保证字符串的完整
!	转义操作符，用于指示其后的一个字符作为一般字符，而不含特殊意义
%	表达式操作符，用在宏调用中，表示将后跟的一个表达式的值作为实参，而不是将表达式本身作为参数
::	宏注释符，用于表示在宏定义中的注释。采用这个符号的注释，在宏展开时不出现
: reg	说明宏定义设定的参数在调用时不可缺少
:= 默认值	设定参数默认值

例如，用宏定义一个自动带有结尾 0 的字符串。

```
asciiz    macro string
           db '&string& ',0
           endm
```

宏定义中的一对“&”伪操作符括起 string，表示它是一个参数。

(1) 定义字符串 ‘This is a example.’，可以采用如下宏调用：

```
asciiz < This is a example.>
```

它产生的宏展开为：

```
1      db 'This is a example.', 0
```

因为字符串中有空格，所以必须采用一对“<>”伪操作符将字符串扩起来。

(2) 定义字符串'0 < Number < 10'，宏调用是：

```
asciiz < 0 !< Number !< 10 >
```

宏展开为：

```
1      db '0 < Number < 10',0
```

字符串中包含“<>”或其他特殊意义的符号，则应该使用转义伪操作符“!”。

(3) 使用表达式值作为参数，如宏调用是：

```
asciiz %(1024-1)
```

宏展开为：

```
1      db '1023', 0
```

3. 宏的伪指令

MASM 设置有若干与宏配合使用的伪指令，旨在增强宏的功能。其中最主要的是局部标号伪指令 LOCAL。

宏定义可被多次调用；每次调用实际上是把替代参数后的宏定义体复制到宏调用的位

置。但是，当宏定义中使用了标号（包括变量定义的名字），同一源程序对它的多次调用就会造成标号的重复定义，汇编将出现语法错误。子程序之所以没有这类问题是因为程序中只有一份子程序代码，子程序的多次调用只是控制的多次转向与返回，某一特定的标号地址是唯一确定的。

所以，如果宏定义体采用了标号，就需要使用局部伪指令 LOCAL 加以说明，格式为：

local 标号列表

其中，标号列表由宏定义体内使用的标号组成，用逗号分隔。这样，每次宏展开时，汇编程序将对其中的标号自动产生一个唯一的标识符（其形式为“??0000”到“??FFFF”），从而避免宏展开后的标号重复。

LOCAL 伪指令只能在宏定义体内使用，而且是宏定义 MACRO 语句之后的第一条语句，两者间也不允许有注释和分号。

例如，编写一个求绝对值的宏，由于具有分支需要采用标号：

```
absol      macro oprd
            local next
            cmp  oprd,0
            jge next
            neg  oprd

next:
            endm                ;这个伪指令要独占一行
```

第 1 次宏调用和宏展开：

```
absol word ptr [bx]
1      cmp word ptr [bx],0
1      jge ??0000
1      neg word ptr [bx]
1      ??0000:
```

第 2 次宏调用和宏展开：

```
absol bx
1      cmp bx,0
1      jge ??0001
1      neg bx
1      ??0001:
```

【例 5-10】 通用寄存器显示程序。

8086 处理器有 8 个 16 位通用寄存器，编写一个显示其内容（以十六进制形式）的子程序，以便随时观察（类似本书提供的子程序 DISPRW）。每个通用寄存器都需要显示 4 位十六进制数，写成宏，转换算法参见例 5-3。

```
                ;宏定义
dreg16 macro reg16
            local dreg1,dreg2
            mov ax,reg16        ;显示 reg16 寄存器
            mov cx,4
            xor bx,bx
dreg1:       rol ax,1
            rol ax,1
            rol ax,1
```

```

rol ax,1
mov dx,ax
and dl,0fh
add dl,30h           ;转化为相应的 ASCII 码值
cmp dl,39h           ;区别 0~9 和 A~F 数码
jbe dreg2
add dl,7
dreg2: mov rd&reg16&[bx+3],dl ;存于对应的字符串
inc bx
cmp bx,cx
jb dreg1
endm
;数据段
rdax db 'AX=0000, '
rdbx db 'BX=0000, '
rdcx db 'CX=0000, '
rddx db 'DX=0000, '
rdsi db 'SI=0000, '
rddi db 'DI=0000, '
rdbp db 'BP=0000, '
rdsp db 'SP=00000',13,10,'$'
;代码段，主程序
mov ax,1357h         ;假设一些数据
mov bx,2468h
mov cx,9abch
mov dx,0defh
mov si,1111h
mov di,2222h
mov bp,sp            ;SP 由系统设置，不要修改
call disprw          ;调用通用寄存器显示子程序
;代码段，子程序
disprw proc           ;8 个 16 位通用寄存器内容显示子程序
push ax              ;入栈保护
push bx
push cx
push dx

push dx
push cx
push bx
dreg16 ax            ;显示 AX
pop bx
dreg16 bx            ;显示 BX
pop cx
dreg16 cx            ;显示 CX
pop dx
dreg16 dx            ;显示 DX

```

```

                dreg16 si                ;显示 SI
                dreg16 di                ;显示 DI
                dreg16 bp                ;显示 BP
                add sp,10                ;获得进入该子程序前的 SP
                dreg16 sp                ;显示 SP
                sub sp,10                ;回复 SP
                mov dx,offset rdax
                mov ah,9
                int 21h
                pop dx                    ;出栈恢复
                pop cx
                pop bx
                pop ax
                ret
disprw          endp

```

宏使用了 AX、BX、CX 和 DX，但没有保护，所以子程序中还要压入堆栈以便恢复，用于显示。子程序首先用了 4 个 PUSH 指令，使得 ESP 减少了 8（即 4×2 ），调用子程序压入 2 字节返回地址，SP 又减少 2，所以 SP 要加 10 才是主程序要显示指令当时的 SP 值。

请大家观察列表文件，如最后的宏调用和展开是：

```

                dreg16 sp                ;显示 SP
1               mov ax,sp
1               mov cx,4
1               xor bx,bx
1  ??000E:      rol ax,1
1               rol ax,1
1               rol ax,1
1               rol ax,1
1               mov dx,ax
1               and dl,0fh
1               add dl,30h
1               cmp dl,39h
1               jbe ??000F
1               add dl,7
1  ??000F:      mov rdx[bx+3],dl
1               inc bx
1               cmp bx,cx
1               jb  ??000E

```

当程序不再需要某个宏定义时，可以把它删除，删除宏定义伪指令 PURGE 的格式为：

purge 宏名表

其中，宏名表是由逗号分隔的需要删除的宏名。宏名一经删除，该标识符就成为未说明的符号串，源程序的后续语句便不能对该名字进行合法的宏调用，但是可以采用这个标识符重新定义其他宏等。

另外，在宏定义体、重复汇编的重复块以及条件汇编的分支代码序列中有时需要配合宏定义退出伪指令 EXITM，它的格式为：

exitm

汇编程序执行 EXITM 指令后立即停止它后面部分的宏展开。

4. 宏与子程序

宏与子程序都可以把一段程序用一个名字定义，简化源程序的结构和设计。一般说来，子程序能实现的功能，用宏也可以实现；但是，宏与子程序却有着本质的不同，主要反映在调用方式上，另外在传递参数和使用细节上也有很多不同。下面从比较的角度简单讨论。

宏调用在汇编时进行程序语句的展开，不需要返回；它仅是源程序级的简化，并不减小目标程序，因此执行速度没有改变。子程序调用在执行时由 CALL 指令转向子程序体，子程序需要执行 RET 指令返回；它还是目标程序级的简化，形成的目标代码较短。但是，子程序需要利用堆栈保存和恢复转移地址、寄存器等，要占用一定的时空开销；特别是当子程序较短时，这种额外开销所占比例较大。

宏调用的参数通过形参、实参结合实现传递，简洁直观、灵活多变。子程序需要利用寄存器、存储单元或堆栈等传递参数。对宏调用来说，参数传递错误通常是语法错误，会由汇编程序发现；而对子程序来说，参数传递错误通常反映为逻辑或运行错误，不易排除。

除此之外，宏与子程序都还具有各自的特点，程序员应该根据具体问题选择使用那种方法。通常，当程序段较短或要求较快执行时，应选用宏；当程序段较长或为减小目标代码时，要选用子程序。

5.4.2 重复汇编

程序中有时需要连续地重复一段相同或者基本相同的语句，这时可以用重复汇编伪指令来完成。重复汇编定义的程序段也是在汇编时展开，并且经常与宏定义配合使用。重复汇编伪指令有 3 个：REPEAT、FOR、FORC（在 MASM 5.x 版本依次是 REPT / IRP / IRPC，它们在后续版本仍然可以使用），都要用 ENDM 结束。重复汇编结构既可以在宏定义体外，也可以在宏定义体内使用。重复汇编的程序段没有名字，不能被调用，但可以使用参数。3 个重复汇编伪指令的不同是如何规定重复次数的具体如下。

1. 按参数值重复伪指令 REPEAT

REPEAT 伪指令的功能是按设定的重复次数连续重复汇编重复体的语句，其格式为：

```
repeat 重复次数    ;重复开始
    .....         ;重复体
endm               ;重复结束
```

【例 5-11】 ASCII 表显示程序。

用重复汇编定义可显示字符的 ASCII 表。

```
                                ;数据段
char=20h                        ;定义第一个可显示字符：空格，其 ASCII 值是 20H
space                            db char
                                repeat 95-1    ;标准 ASCII 表，有 95 个可显示字符
                                    char=char+1
                                    db char
                                endm
                                ;代码段
                                mov cx,95
```

```

                                mov bx,offset space
again:                          mov dl,[bx]
                                mov ah,2
                                int 21h
                                inc bx
                                loop again

```

重复汇编的展开有 94 个语句，每个 CHAR 常量增加 1：

```

1          db char

```

由于字符表中包含“\$”字符，所以不能使用 9 号 DOS 字符串显示的功能调用。

2. 按参数个数重复伪指令 FOR

FOR 伪指令的功能是按实参表的参数个数连续重复汇编重复体的语句。实参表用尖括号括起，参数以逗号分隔，按照参数从左到右的顺序，每一次的重复把重复体中的形参用一个实参取代。它的格式为：

```

for 形参, <实参表>
.....          ;;重复体
endm

```

IA-32 处理器有保护所有通用寄存器的指令 PUSHAD，对应的恢复所有寄存器的指令是 POPAD。有时，子程序只需要保护常用寄存器，可以如下编写：

```

for regad, <ax,bx,cx,dx>
    push regad
endm

```

汇编后产生如下代码：

```

1          push ax
1          push bx
1          push cx
1          push dx

```

3. 按参数字符个数重复伪指令 FORC

FORC 伪指令的功能是按字符串的字符个数连续重复汇编重复体的语句。字符串用或不用尖括号括起，按照字符从左到右的顺序，每一次的重复把重复体中的形参用一个字符取代。它的格式为：

```

forc 形参,字符串          ;;或 FORC 形参, <字符串>
.....          ;;重复体
endm

```

例如，子程序最后恢复常用寄存器的系列语句可以如下编写：

```

forc regad,dcba
    pop e&regad&x
endm

```

汇编后产生如下代码：

```

1          pop dx
1          pop cx
1          pop bx
1          pop ax

```

5.4.3 条件汇编

条件汇编伪指令根据某种条件确定是否汇编某段语句序列，它与高级语言的条件编译命令类似。条件汇编伪指令的一般格式是：

```
ifxx 表达式          ;;条件满足，汇编分支语句体 1
    分支语句体 1
[ else                ;;条件不满足，汇编分支语句体 2
    分支语句体 2 ]
endif                ;;条件汇编结束
```

其中，IF 后跟的 xx 表示组成条件汇编伪指令的其他字符，如表 5-4 所示。如果表达式表示的条件满足，汇编程序将汇编分支语句体 1 中的语句；不满足条件，分支语句体 1 不被汇编。若存在可选的 ELSE 伪指令，分支语句体 2 中的语句将在不满足条件时被汇编。

表 5-4 条件汇编伪指令

格 式	功 能 说 明
IF 表达式	汇编程序求出表达式的值，此值不为 0，则条件满足
IFE 表达式	汇编程序求出表达式的值，此值为 0，则条件满足
IFDEF 符号	符号已定义（内部定义或声明外部定义），则条件满足
IFNDEF 符号	符号未定义，则条件满足
IFB <形参>	用在宏定义体。如果宏调用没有用实参替代该形参，则条件满足
IFNB <形参>	用在宏定义体。如果宏调用用实参替代该形参，则条件满足
IFIDN <字符串 1>,<字符串 2>	字符串 1 与字符串 2 相同则条件满足；区别大小写
IFDIF <字符串 1>,<字符串 2>	字符串 1 与字符串 2 不相同则条件满足；区别大小写
IFIDNI <字符串 1>,<字符串 2>	字符串 1 与字符串 2 相同则条件满足；不区别大小写
IFDIFI <字符串 1>,<字符串 2>	字符串 1 与字符串 2 不相同则条件满足；不区别大小写

1. IF/IFE 伪指令

IF/IFE 伪指令根据表达式是否成立决定是否汇编。表达式采用汇编语言的关系运算符，它们是：EQ（相等）、NE（不相等）、GT（大于）、LT（小于）、GE（大于等于）、LE（小于等于）。关系表达式用 -1（非 0）表示成立（真），用 0 表示不成立（假）。

例如，定义一个元素个数不超过 100 的数组：

```
pdata    macro num
          if num lt 100      ;如果 num < 100，则汇编如下语句
              db num dup (?)
          else                ;否则，汇编如下语句
              db 100 dup (?)
          endif
        endm
```

如果宏调用的实参小于等于 100，例如：

```
pdata 12
```

则汇编 db num dup (?)语句：

```
db 12 dup(?)
```

如果宏调用的实参大于或等于 100，例如：

```
pdata 102
```

则汇编 db 100 dup (?)语句:

```
db 100 dup(?)
```

2. IFDEF/IFNDEF 伪指令

IFDEF/IFNDEF 伪指令根据指定的标识符是否被定义决定是否汇编。

例如，程序中有时使用本书提供的输入输出子程序库很方便，如果用符号 iolib 表示，则当发现定义了该符号就汇编该包含语句，如下所示：

```
ifdef iolib                ;当定义有 iolib 符号时，汇编如下语句
    include io.inc
endif
```

这样，在源程序开始书写一个语句，就可以使用本书的子程序库了：

```
iolib=1                    ;定义 iolib 符号
```

3. IFB/IFNB 伪指令

IFB/IFNB 伪指令用在宏定义中，根据宏调用时是否用实参替代形参进行条件汇编。

例如，编写宏 MAXNUM，计算 3 个以内的数（形参）中的最大值，并将结果送入 AX 寄存器。实际参加比较的数应该有一个，所以第一个参数设定为不可省略 (: req)。如果只有两个参数，则只需比较一次，后一个比较的代码不用展开。宏定义体中判断后两个实参是否为空而汇编相应的比较代码。

```
maxnum    macro var1:req,var2,var3
            local maxnum1,maxnum2
            mov ax,var1
            ifnb <var2>      ;;当有 VAR2 实参时，汇编如下语句
                cmp ax,var2
                jge maxnum1
                mov ax,var2
            endif
maxnum1:
            ifnb <var3>      ;;当有 VAR3 实参时，汇编如下语句
                cmp ax,var3
                jge maxnum2
                mov ax,var3
            endif
maxnum2:
            endm
```

如果第 1 次宏调用是：

```
maxnum bx
```

则第 1 次宏汇编的结果是：

```
mov ax,bx
```

如果第 2 次宏调用是：

```
maxnum 3,5
```

则第 2 次宏汇编的结果是：

```

mov ax,3
cmp ax,5
jge ??0002
mov ax,5

```

??0002:

如果第 3 次宏调用是:

```
maxnum 3,6,9
```

则第 3 次宏汇编的结果是:

```

mov ax,3
cmp ax,6
jge ??0004
mov ax,6

```

??0004:

```

cmp ax,9
jge ??0005
mov ax,9

```

??0005:

4. IFIDN/IFDIF 和 IFIDNI/IFDIFI 伪指令

IFIDN/IFDIF 和 IFIDNI/IFDIFI 伪指令根据两个字符串是否相等进行条件汇编，前一对区别字母大小写，后一对不区别字母大小写。

【例 5-12】 格式化显示程序。

用不同的编码解释同一个计算机内部代码，其含义不同，反映在显示输出上也应该不同。这就像 C 语言的 printf 函数一样，它支持多种格式符输出。利用条件汇编编写一个类似的宏，根据不同的格式符号，汇编不同的语句，实现对应格式显示。

```

;宏定义
printf macro format,var
mov al,var
ifidni <format>,<b>
call dispbpb ;二进制显示用格式符“b”
exitm ;不再进行宏展开
endif
ifidni <format>,<x>
call disphb ;十六进制显示用格式符“x”
exitm
endif
ifidni <format>,<d>
call dispsib ;十进制显示用格式符“d”
exitm
endif
ifidni <format>,<c>
call dispcc ;字符显示用格式符“c”
endif
endm
;数据段

```

```

var          db 01100100b
             ;代码段
             printf b,var          ;二进制形式显示: 01100100
             call dispCrLf         ;回车换行 (用于分隔)
             printf x,var          ;十六进制形式显示: 64
             call dispCrLf         ;回车换行 (用于分隔)
             printf d,var          ;十进制形式显示: 100
             call dispCrLf         ;回车换行 (用于分隔)
             printf c,var          ;字符显示: d

```

本例程序需要使用本书提供的二进制显示 DISPBB、十六进制显示 DISPHB、有符号十进制显示 DISPSIB、字符显示 DISPC 和回车换行 DISPCRLF 子程序, 所以开始要包含 IO.INC 文件。根据本章所学, 大家也可以自己创建这些子程序。

习 题 5

5.1 简答题

- (1) 指令“CALL BX”采用了指令的什么寻址方式?
- (2) 为什么 MASM 要求使用 PROC 定义子程序?
- (3) 为什么特别强调为子程序加上必要的注释?
- (4) 参数传递的“传值”和“传址”有什么区别?
- (5) 子程序采用堆栈传递参数, 为什么要特别注意堆栈平衡问题?
- (6) INCLUDE 语句和 INCLUDELIB 语句有什么区别?
- (7) 什么是子程序库?
- (8) 调用宏时没有为形参提供实参会怎样?
- (9) 宏定义体中的标号为什么要用 LOCAL 伪指令声明?
- (10) 条件汇编不成立的语句会出现在可执行文件中吗?

5.2 判断题

- (1) 过程定义 PROC 是一条处理器指令。
- (2) CALL 指令的执行并不影响堆栈指针 SP。
- (3) CALL 指令本身不能包含子程序的参数。
- (4) CALL 指令用在调用程序中, 如果被调用程序中也有 CALL 指令, 说明出现了嵌套。
- (5) 子程序需要保护寄存器, 包括保护传递入口参数和出口参数的通用寄存器。
- (6) 利用 INCLUDE 包含的源文件实际上只是源程序的一部分。
- (7) 宏调用与子程序调用一样都要使用 CALL 指令实现。
- (8) 宏定义可以与子程序一样, 书写于主程序之后。
- (9) 重复汇编类似宏汇编, 需要先定义后调用。
- (10) 条件汇编并不像条件转移指令那样使用处理器状态标志作为条件。

5.3 填空题

- (1) 指令“RET il6”的功能相当于“RET”指令和“ADD SP, _____”组合。
- (2) 例 5-1 程序中的 RET 指令, 如果用 POP BP 指令和 JMP BP 指令替换, 此时 BP 内容是_____。

- (3) 子程序的参数传递主要有 3 种，它们是_____、_____和_____。
- (4) 数值 10 在计算机内部用二进制“1010”编码表示，用十六进制表达是：_____。如果将该编码加 37H，则为_____，它是字符_____的 ASCII 码值。
- (5) 利用堆栈传递子程序参数的方法是固定的，例如寻址堆栈段数据的寄存器是_____。
- (6) MASM 汇编语言中，声明一个共用的变量应使用_____伪指令；而使用外部变量要使用_____伪指令声明。
- (7) 过程定义开始是“TEST PROC”语句，则过程定义结束的语句是_____。宏定义开始是“DISP MACRO”语句，则宏定义结束的语句是_____。
- (8) 一个宏定义开始语句“WriteChar MACRO CHAR:REQ”，则宏名是_____，参数有_____个，并且使用“:REQ”说明该参数_____。
- (9) 实现“DB 20 DUP(20H)”语句的功能也可以使用重复汇编，第 1 个语句是_____，第 2 个语句是“DB 20H”，第 3 个语句是_____。
- (10) 条件汇编语句“IF NUM LT 100”中的 LT 表示_____；该语句需要配合_____语句结束条件汇编。

5.4 如下子程序完成对 CX 个元素的数组（由 BX 指向其首地址）的求和，通过 DX 和 AX 返回结果，但程序有错误，请改正。

```
crazy      PROC
            push ax
            xor ax,ax
            xor dx,dx
again:      add ax,[bx]
            adc dx,0
            add bx,2
            loop again
            ret
            ENDP crazy
```

5.5 请按如下说明编写子程序。

子程序功能：把用 ASCII 码表示的两位十进制数转换为压缩 BCD 码

入口参数：DH=十位数的 ASCII 码，DL=个位数的 ASCII 码

出口参数：AL=对应 BCD 码

5.6 乘法的非压缩 BCD 码调整指令 AAM 执行的操作是： $AH \leftarrow AL \div 10$ 的商， $AL \leftarrow AL \div 10$ 的余数。利用 AAM 可以实现将 AL 中的 100 内数据转换为 ASCII 码，程序如下：

```
xor ah,ah
aam
add ax,3030h
```

利用这段程序，编写一个显示 AL 中数值（0~99）的子程序。

5.7 编写一个源程序，在键盘上按一个键，将其返回的 ASCII 码值显示出来，如果按下 ESC 键（对应 ASCII 码是 1BH）则程序退出。请调用书中的 HTOASC 子程序。

5.8 编写一个子程序，它以二进制形式显示 AL 中 8 位数据，并设计一个主程序验证。

5.9 参考例 5-4，编写一个仅实现输出 8 位即 -128~+127 之间数据的子程序，并设计一个主程序验证。

5.10 参考例 5-6，编写实现 8 位无符号整数输入的子程序，并设计一个主程序验证。

5.11 编写一个计算字节校验和的子程序。所谓“校验和”，是指不记进位的累加，常用于检查信息的正确性。主程序提供入口参数，有数据个数和数据缓冲区的首地址。子程序回送求和结果这个出口参数。

5.12 编制 3 个子程序，把一个 16 位二进制数用 4 位十六进制形式在屏幕上显示出来，分别运用如下 3 种参数传递方法，并配合 3 个主程序验证它。

(1) 采用 AX 寄存器传递这个 16 位二进制数。

(2) 采用 temp 变量传递这个 16 位二进制数。

(3) 采用堆栈方法传递这个 16 位二进制数。

5.13 设计一个从低地址到高地址逐字节显示某个主存区域内容的子程序 DISPMEM。入口参数：AX=主存偏移地址，CX=字节个数（主存区域的长度）。同时编写一个主程序进行验证。

5.14 数据输入输出程序。

使用有符号十进制数据输入（例 5-6）、求平均值（例 5-7）以及输出子程序（例 5-4），编程实现从键盘输入 10 个数据并将它们的平均值输出。

(1) 编写主程序文件：定义必要的变量和交互信息，调用子程序输入 10 个数据、求平均值，然后输出。

(2) 编写子程序文件：包括 3 个子程序的过程定义。

(3) 说明进行模块连接的开发过程并上机实现。

(4) 将子程序文件形成一个子程序库，说明开发过程并上机实现。

5.15 区别如下概念：宏定义、宏调用、宏指令、宏展开、宏汇编。

5.16 宏如何定义和调用？

5.17 宏结构和子程序在应用中有什么不同，如何选择采用何种结构？

5.18 编写一个宏 SWAP，参数是两个 16 位寄存器或存储器操作数，宏定义体实现两个操作数交换位置，包括两个都是存储器操作数的情况。

5.19 定义一个使用逻辑指令的宏 LOGICAL。

(1) 用它代表 4 条逻辑运算指令：AND/OR/XOR/TEST，可以使用 3 个形式参数，并给一个宏调用以及对应宏展开的例子。

(2) 必要时做一点修改，使该宏能够把 NOT 指令包括进去，给一个使用 NOT 指令的宏调用以及对应宏展开的例子。

5.20 有一个宏定义：

```
defstr      macro name,num,string
name&num    db '&string&',0
endm
```

给出如下宏调用的宏展开：

(1) defstr msg,4,< Chapter 4: Program Structure >

(2) defstr msg,5,< Chapter 5: Procedure Programming >

5.21 定义一个宏 movestr strN,dstr,sstr，它将 strN 个字符从一个字符区 sstr 传送到另一个字符区 dstr。

假设数据段定义如下缓冲区，请使用上述宏的调用实现 STRING1 到 STRING2 的传送。

```
string1     db 'In a major matter, no details are small.',0
string2     db sizeof string1 dup(0)
```


5.22 利用重复汇编方法定义一个数据区，数据区有 100 个字，每个字的高字节部分依次是 2、4、6、…、200，低字节部分都是 0。

5.23 利用宏结构完成以下功能：如果名为 COUNT 的数据大于 5，指令“ADD AX,AX”将汇编 10 次，否则什么也不汇编。

5.24 用宏结构实现宏指令 FINSUM，它比较两个数 VARX 和 VARY，若 $\text{VARX} \geq \text{VARY}$ ，则执行 $\text{SUM} = \text{VARX} + 8 \times \text{VARY}$ ，否则执行 $\text{SUM} = 4 \times \text{VARX} + \text{VARY}$ 。

第 6 章 32 位指令及其编程

前 5 章介绍了基于 Intel 8086 处理器指令系统的汇编语言程序设计。从本章开始引申到 Intel 80x86 系列处理器。读者在了解 Intel 80x86 处理器的发展概况基础上,将熟悉 32 位整型指令的运行环境及特点,学习 DOS 平台的 32 位指令编程。

6.1 Intel 80x86 处理器

美国 Intel 公司是目前世界上最有影响的处理器生产厂家,也是世界上第一个微处理器芯片的生产厂家。Intel 80x86 系列处理器一直是个人计算机的主流处理器。

6.1.1 16 位 80x86 处理器

1971 年,Intel 公司生产的 4 位处理器芯片 Intel 4004 宣告了微型计算机时代的到来。1972 年,Intel 公司开发了 8 位处理器 Intel 8008 芯片;1974 年,生产了 Intel 8080;1977 年,Intel 公司将 8080 及其支持电路集成在一块集成电路芯片上,形成了性能更高的 8 位处理器 8085。1978 年,Intel 公司在其 8 位处理器的基础上,陆续推出了 16 位结构的 8086、8088 和 80286 等处理器;它们在 IBM PC 系列机中获得广泛应用,被称为 16 位 80x86 处理器。

1. 8086

1978 年,Intel 推出 16 位 8086 处理器,这是该公司生产的第一个 16 位结构处理器芯片。8086 芯片的对外引脚共有 40 个,其中包括 16 位数据总线、20 位地址总线,支持 1MB 主存容量、5MHz 时钟频率。8086 指令系统是 Intel 80x86 系列处理器的 16 位基本指令集。

为了方便与当时的 8 位外部设备连接,1979 年,Intel 推出了被称为“准 16 位处理器”的 Intel 8088。8088 只是将外部数据总线设计为 8 位,内部仍保持 16 位结构,指令系统等都与 8086 相同。随后的 Intel 80186 和 Intel 80188 则是分别以 8086 和 8088 为核心并配以支持电路构成的芯片,它们在 8086 指令系统的基础上新增了若干条实用指令,涉及堆栈操作、输入输出指令、移位指令、乘法指令、支持高级语言的指令。

2. 80286

1982 年,Intel 推出仍为 16 位结构的 Intel 80286 处理器,但地址总线扩展为 24 位,即主存储器具有 16MB 容量。Intel 80286 设计有与 8086 工作方式一样的实方式 (Real Mode),新增保护方式 (Protected Mode)。在实方式下,Intel 80286 相当于一个快速 8086。在保护方式下,Intel 80286 提供了存储管理、保护机制和多任务管理的硬件支持。为了支持保护方式,80286 引入了系统指令,为操作系统等核心程序提供处理器控制功能。

6.1.2 IA-32 处理器

IBM PC 系列机的广泛应用推动了处理器芯片的生产。Intel 公司在推出 32 位结构的 80386 处理器后,明确宣布 Intel 80386 芯片的指令集结构 ISA (Instruction Set Architecture) 被确定

为以后开发的 80x86 系列处理器的标准，称为 Intel 32 位结构：IA-32（Intel Architecture-32）。现在，Intel 公司的 80386、80486 及 Pentium 各代处理器被通称为 IA-32 处理器或 32 位 80x86 处理器。

1. 80386

1985 年，Intel 80x86 处理器进入第 3 代 80386 时代。Intel 80386 处理器采用 32 位结构，数据总线 32 位，地址总线也是 32 位，可寻址 4GB 主存，时钟频率有 16MHz、25MHz 和 33MHz。IA-32 指令系统在兼容原 16 位 80286 指令系统基础上，全面升级为 32 位，新增了位操作、条件设置等指令。

80386 除保持与 Intel 80286 兼容外，又提供了虚拟 8086 工作方式（Virtual 8086 Mode）。虚拟 8086 方式是在保护方式下的一种特殊状态，类似 8086 工作方式但又接受保护方式的管理，能够模拟多个 8086 处理器。32 位 PC 的 Windows 操作系统采用保护方式，其 MS-DOS 命令行（环境）就是虚拟 8086 方式，而早期采用的 DOS 操作系统是以实方式为基础建立的。

2. 80486

1989 年，Intel 公司出品 80486 处理器。它内部集成了 120 万个晶体管，最初的时钟频率为 25MHz，很快发展到 33MHz 和 50MHz。从结构上来说，Intel 80486 把 Intel 80386 处理器与 80387 数学协处理器和 8KB 高速缓冲存储器（Cache）集成在一个芯片上，使处理器的性能大大提高。

传统上，CPU 主要是整数处理器。为了协助处理器处理浮点数据（实数），Intel 设计有数学协处理器，后被称为浮点处理单元 FPU（Floating-Point Unit）。配合 8086 和 8088 整数处理器的数学协处理器是 8087，配合 80286 的是 80287，80386 采用 80387。而从 Intel 80486 开始，FPU 已经被集成到一个处理器当中，IA-32 处理器的指令系统就包含了浮点指令，能够直接支持对浮点数据的处理。

3. Pentium 系列

Pentium 芯片原来应该被称为 80586 处理器，因为数字很难进行商标版权保护而特意取名。其实，Pentium 源于希腊文“pente”（数字 5），加上后缀-ium（化学元素周期表中命名元素常用的后缀）变化而来的。同时，Intel 公司为其取了一个响亮的中文名称：奔腾，并进行了商标注册，形成了系列产品。

Intel 公司于 1993 年成功制造 Pentium，于 1995 年正式推出 Pentium Pro（原来被称为 P6，中文名称为“高能奔腾”）。在处理器结构上，Pentium 主要引入了超标量（Superscalar）技术，Pentium Pro 主要采用动态执行技术来提升处理器性能，增加了若干整数指令，完善了浮点指令。

前面所述的各代 IA-32 处理器都新增有若干实用指令，但非常有限。为了顺应微机向多媒体和通信方向发展，Intel 公司及时在其处理器中加入了多媒体扩展 MMX（MutliMedia eXtension）技术。MMX 技术于 1996 年正式公布，在 IA-32 指令系统中新增了 57 条整数运算多媒体指令，可以用这些指令对图像、音频、视频和通信方面的程序进行优化，使微机对多媒体的处理能力较原来有了大幅度提升。MMX 指令应用于 Pentium 处理器就是 Pentium MMX（多能奔腾）。MMX 指令应用于 Pentium Pro 处理器就是 Pentium II，它于 1997 年推出。

1999 年, 针对国际互联网和三维多媒体程序的应用要求, Intel 在 Pentium II 的基础上新增了 70 条 SSE(Streaming SIMD Extensions)指令(原称为 MMX-2 指令), 开发了 Pentium III。SSE 指令侧重于浮点单精度多媒体运算, 极大地提高了浮点 3D 数据的处理能力。SSE 指令类似于 AMD 公司发布的“3D Now !”指令。由于这些多媒体指令具有显著的单指令多数据 SIMD (Single Instruction Multiple Data) 处理能力, 即一条指令可以同时进行了多组数据的操作, 现在统称为 SIMD 指令。

2000 年 11 月, Intel 公司推出 Pentium 4, 新增 76 条 SSE2 指令集, 侧重于增强浮点双精度多媒体运算能力。2003 年的新一代 Pentium 4 处理器又新增了 13 条 SSE3 指令, 用于补充完善 SIMD 指令集。

6.1.3 Intel 64 处理器

随着互联网、多媒体、3D 视频等的发展, 信息时代的应用对计算机性能提出了越来越高的要求, 32 位单核处理器已不能适应这一要求。

1. Intel 64 结构

一直以来, 80x86 处理器的更新换代都保持与早期处理器的兼容, 以便继续使用现有的软硬件资源。但是, Intel 公司迟迟不愿将 80x86 处理器扩展为 64 位, 这给了 AMD 公司一个机会。AMD 公司是生产 IA-32 处理器兼容芯片的厂商, 是 Intel 公司最主要的竞争对手。AMD 公司的 IA-32 兼容处理器, 其价格低于 Intel 芯片, 但性能却没有超越对应的 Intel 芯片。于是, AMD 公司于 2003 年 9 月率先推出支持 64 位、兼容 80x86 指令集结构的 Athlon 64 处理器 (K8 核心), 将桌面 PC 引入了 64 位领域。

2005 年, 在 PC 用户对 64 位技术的企盼和 AMD 公司 64 位处理器的压力下, Intel 公司推出了扩展存储器 64 位技术 Intel EM64T (Intel Extended Memory 64 Technology)。EM64T 技术是 IA-32 结构的 64 位扩展, 首先应用于支持超线程技术的 Pentium 4 终极版 (支持双核技术) 和 6xx 系列 Pentium 4 处理器。随着 EM64T 技术的出现, IA-32 指令系统也扩展成为 64 位, 后来被称为 Intel 64 结构。这之后的 Pentium 4 处理器以及 Pentium E 系列多核处理器、酷睿 (Core) 2 和酷睿 i 系列多核处理器等都支持 Intel 64 结构。

Intel 64 位结构为软件提供了 64 位线性地址空间, 支持 40 位物理地址空间。IA-32 处理器支持保护方式 (含虚拟 8086 方式)、实地址方式和系统管理 SMM 方式, Intel 64 位结构则引入了一个新的工作方式: 32 位扩展工作方式 IA-32e。IA-32e 除有一个运行 32 位和 16 位软件的兼容方式, 还有一个 64 位方式。在 64 位工作方式, 允许 64 位操作系统运行存取 64 位地址空间的应用程序, 还可以存取 8 个附加的通用寄存器、8 个附加的 SIMD 多媒体寄存器、64 位通用寄存器和 64 位指令指针等。

2. 多核技术

单纯提高时钟频率等传统的增加处理器复杂度的方法不仅很难提升处理器性能, 还带来功耗剧增、发热量巨大的问题。在这种情况下, 多核 (Multi-core) 技术应运而生。多核处理器是在一个集成电路芯片上制作了两个或多个处理器执行核心, 依靠多个处理器核心相互协作同时执行多个程序线程来提升性能。基于不同的处理器内部结构, Intel 也推出了多款多核处理器, 如 Intel 奔腾 E 系列多核处理器、酷睿 2 和酷睿 i 系列多核处理器。

另一方面，Intel 公司继续丰富了 SSE 系列指令集，酷睿 2 补充了 SSE3 指令（即 32 条 SSSE3 指令），后又推出增加了 54 条指令的 SSE4 指令集。其中 47 条指令引入到了 Intel 面向服务器领域的至强（Xeon）5400 系列和酷睿 2 至尊版 QX9650，被称为 SSE4.1 指令，致力于提升多媒体、3D 处理等的性能；其余 7 条指令被称为 SSE4.2 指令。

Intel 公司充分利用集成电路生产的先进技术和处理器结构的革新技术，推出了多种 Intel 80x86 系列处理器芯片。就目前的发展来看，Intel 公司正在利用单芯片多处理器技术生产双核、四核等多核处理器，并推广支持 64 位处理器和 64 位软件的微机。

6.2 32 位指令运行环境

相对 16 位 8086 处理器来说，32 位 IA-32 处理器在很多方面都有提升，对于底层程序员而言，我们将主要了解寄存器、存储器模型、寻址方式以及指令代码等相关知识。

6.2.1 32 位寄存器

IA-32 处理器除通用寄存器与指令指针、段寄存器、标志寄存器外，还有控制寄存器、系统地址寄存器、调试寄存器和测试寄存器，共 7 类。应用程序主要涉及前 4 类寄存器（如图 6-1 所示），只有系统程序才会用到所有寄存器。

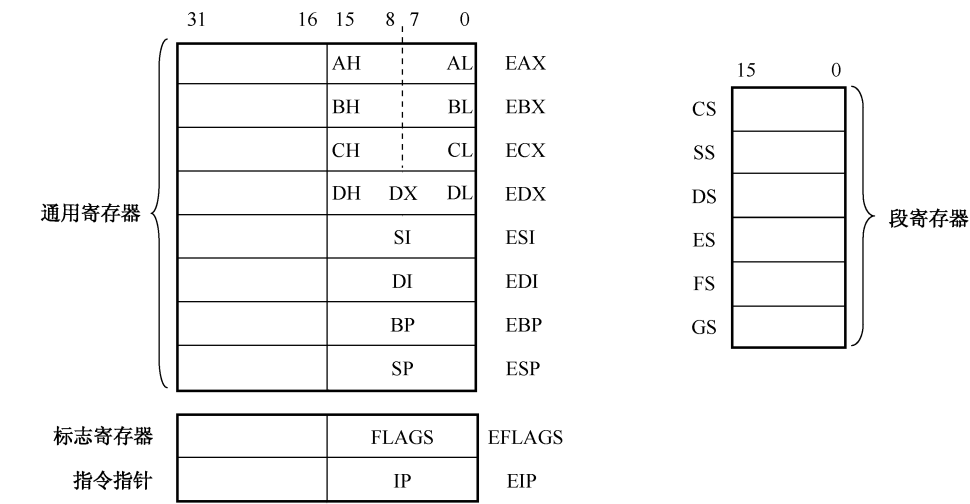


图 6-1 IA-32 常用寄存器

1. 通用寄存器

对于 32 位 IA-32 处理器来说，其通用寄存器自然是 32 位，它们是在原来 8086 支持的 16 位通用寄存器的基础上扩展而成的，加上表达扩展含义的字母 E（Extended）就是现在的 8 个 32 位通用寄存器：EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP。同时，它们仍然可以独立使用其 16 位，即 AX、BX、CX、DX、SI、DI、BP 和 SP，表示相应 32 位通用寄存器低 16 位部分。同样，其中前 4 个通用寄存器 AX、BX、CX 和 DX 还可以进一步分成高字节 H（High）和低字节 L（Low）两部分，这样又有了 8 个 8 位通用寄存器：AH 和 AL、BH 和 BL、CH 和 CL、DH 和 DL。

编程中可以使用 32 位寄存器（如 ESI），也可以只使用其低 16 位部分（名称中去掉字母 E，如 SI，多用在 16 位平台和操作 16 位数据时）。对其中前 4 个 32 位通用寄存器（如 EAX），可以使用全部 32 位：D31~D0（EAX），可以使用低 16 位：D15~D0（AX），还可以将低 16 位再分成两个 8 位使用：D15~D8（AH）和 D7~D0（AL）。注意，指令在操作这些 16 位寄存器时，相应的 32 位寄存器的高 16 位不受影响；操作这些 8 位寄存器时，相应的 16 位和 32 位寄存器的其他位也不受影响。这样，Intel 80x86 处理器一方面保持了相互兼容，另一方面可以方便地支持 8 位、16 位和 32 位操作。

2. 标志寄存器

IA-32 处理器在原来 16 位标志寄存器基础上，各代 80x86 处理不断增加，形成了 32 位 EFLAGS 标志寄存器，如图 6-2 所示。EFLAGS 标志寄存器包含一组状态标志、一个控制标志和一组系统标志，其初始状态为 000000002H（也就是 D1 位为 1、其他位全部为 0，其中 1、3、5、15 和 22~31 位被保留，软件不需要使用它们或依赖于这些位的状态）。

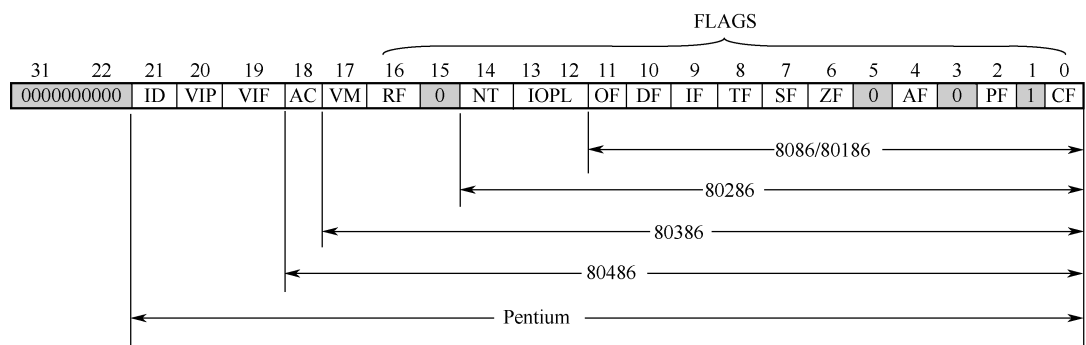


图 6-2 标志寄存器 EFLAGS

原来 8086 具有的标志没有变化，80286 以后新增的标志主要用于处理器控制，通常不在应用程序中使用，如表 6-1 所示。

表 6-1 80x86 新增标志

英文名称	中文名称及作用
NT (Nested Task)	任务嵌套标志。若 NT=1，表示当前执行的任务，嵌套于另一个任务中，待执行完毕时应返回原来的任务
IOPL (I/O Privilege Level)	I/O 特权层标志。共 2 位，编码表示 4 个特权级别，用来指定任务的 I/O 操作处于 4 个特权级别的哪一层
VM (Virtual 8086 Mode)	虚拟 8086 方式。在处于保护方式时，如果使 VM=1 置位，进入虚拟 8086 方式
RF (Resume Flag)	恢复标志。这个标志与调试寄存器一起使用
AC (Alignment Check)	对齐检测标志。设置是否在存储器访问时进行数据对齐检测
VIF (Virtual Interrupt Flag)	虚拟中断标志。IF 中断允许标志的虚拟影像，与 VIP 连用
VIP (Virtual Interrupt Pending)	虚拟中断挂起标志。此标志置位指示有一个中断被挂起
ID (Identification Flag)	CPU 识别标志。程序如果能够置位和复位这个标志位，则表示该微处理器支持 CPU 识别指令 CPUID

3. 指令指针和段寄存器

在 IA-32 处理器中, 指令指针寄存器也扩展为 32 位, 名称为 EIP。

除原有的代码段寄存器 CS、堆栈段寄存器 SS、数据段寄存器 DS 和附加段寄存器 ES 外, IA-32 处理器增加了两个数据段性质的 16 位段寄存器 FS 和 GS。

4. 其他寄存器

理解处理器工作原理和编写系统程序时都需要了解系统专用寄存器。这些寄存器主要用于保护方式下系统程序控制和“特权”指令操作。例如, 指向系统中特殊段的系统地址寄存器: 全局描述符表寄存器 GDTR、中断描述符表寄存器 IDTR、局部描述符表寄存器 LDTR 和任务状态段寄存器 TR, 它们用于支持存储管理。还有 5 个用于保存系统中所有任务机器状态的控制寄存器 CRn ($n=0\sim4$) 和 8 个用于内部调试的调试寄存器 DRn ($n=0\sim7$) 等。

此外, IA-32 处理器在浮点处理单元 FPU (Floating-Point Unit) 中还有浮点寄存器和支持多媒体指令的多媒体寄存器, 这些将在第 9 章详述。

6.2.2 存储器模型

IA-32 处理器支持 4GB 存储器, 物理地址空间是 $0\sim2^{32}-1$, 用 8 位十六进制数表达物理地址: 00000000H~FFFFFFFFH。为了有效地使用存储器, 几乎所有操作系统和核心程序都具有存储管理功能, 即动态地为程序分配存储空间的能力。IA-32 处理器内部设计有存储管理单元, 提供分段和分页管理机制, 以存储模型形式供程序员使用主存储器。

1. 存储模型

访问存储管理单元后, 程序并不直接寻址物理存储器。IA-32 处理器提供 3 种存储模型 (Memory Model), 用于程序访问存储器。

(1) 平展存储模型 (Flat Memory Model)

平展存储模型下, 对程序来说, 存储器是一个连续的地址空间, 被称为线性地址空间。程序需要的代码、数据和堆栈都包含在这个地址空间中。线性地址空间也以字节为基本存储单位, 即每个存储单元保存一字节、具有一个地址, 这个地址被称为线性地址 (Linear Address)。IA-32 处理器支持的线性地址空间是 $0\sim2^{32}-1$ (4GB 容量)。

(2) 段式存储模型 (Segmented Memory Model)

段式存储模型下, 对程序来说, 存储器由一组独立的地址空间组成, 这个地址空间被称为段 (Segment)。通常, 代码、数据和堆栈处于分开的段中。程序利用逻辑地址 (Logical Address) 寻址段中的每个字节单元, 每个段都可以达到 4GB。

在处理器内部, 所有的段都被映射到线性地址空间。程序访问一个存储单元时, 处理器会将逻辑地址转换成线性地址。使用段式存储模型, 是为了提高程序的可靠性。例如, 将堆栈安排在分开的段中, 可以防止堆栈区域增加时侵占代码或数据空间。

(3) 实地址存储模型 (Real-address Mode Memory Model)

实地址存储模型是 8086 处理器的存储模型。IA-32 处理器之所以支持这种存储模型, 是为了兼容原为 8086 处理器编写的程序。实地址存储模型是段式存储模型的特例, 其线性地址空间最大为 1MB, 由最大为 64KB 的多个段组成。

2. 工作方式

编写程序时，程序员还需要明确处理器执行代码的工作方式，工作方式决定可以使用的指令和存储模型。IA-32 处理器支持 3 种基本的工作方式（操作模式）：保护方式、实地址方式和系统管理方式。

（1）保护方式（Protected Mode）

保护方式是 IA-32 处理器固有的工作方式。在保护方式下，IA-32 处理器能够完成其全部功能，可以充分利用其强大的段页式存储管理和特权与保护能力。保护方式下，IA-32 处理器可以使用全部 32 条地址总线，可寻址 4GB 物理存储器。

IA-32 处理器从硬件上实现了特权的 management 功能，方便操作系统使用，为不同程序设置了 4 个特权层（Privilege Level）：0~3（数值越小表示特权级别越高，所以特权层 0 级别最高）。例如，特权层 0 用于操作系统中负责存储管理、保护和存取控制部分的核心程序；特权层 1 用于操作系统；特权层 2 可专用于应用子系统（数据库管理系统、办公自动化系统和软件开发环境等）；应用程序使用特权层 3。这样，系统核心程序、操作系统、其他系统软件以及应用程序，可以根据需要分别处于不同的特权层而得到相应的保护。当然，在没有必要时不一定使用所有的特权层。例如，在微机中，Windows 操作系统处于特权层 0，应用程序则处于特权层 3。

保护方式具有直接执行实地址 8086 软件的能力，这个特性被称为虚拟 8086 方式（Virtual-8086 mode）。虚拟 8086 方式并不是处理器的一种工作方式，只是提供了一种在保护方式下类似实地址方式的运行环境，如 Windows 中的 MS-DOS 运行环境。

处理器在保护方式下工作时，可以使用平展或段式存储模型；在虚拟 8086 方式下工作时，只能使用实地址存储模型。

（2）实地址方式（Real-address Mode）

通电或复位后，IA-32 处理器进入实地址工作方式（简称实方式）。它实现了与 8086 相同的程序设计环境，但有所扩展。实地址方式下，IA-32 处理器只能寻址 1MB 物理存储器空间，每个段最大不超过 64KB；可以使用 32 位寄存器、32 位操作数和 32 位寻址方式，相当于可以进行 32 位处理的快速 8086。

实地址方式具有最高特权，而虚拟 8086 方式处于最低特权层 3 下。所以，虚拟 8086 方式的程序都要经过保护方式所确定的所有保护性检查。

实地址工作方式只支持实地址存储模型。

（3）系统管理方式（System Management Mode）

系统管理方式为操作系统和核心程序提供节能管理和系统安全管理等机制。进入系统管理方式后，处理器首先保存当前运行程序或任务的基本信息，然后切换到一个分开的地址空间，执行系统管理相关的程序。退出 SMM 方式时，处理器将恢复原来程序的状态。

处理器在系统管理方式下切换到的地址空间，称为系统管理 RAM，使用类似实地址的存储模型。

3. 逻辑地址

不论是何种存储模型，程序员都采用逻辑地址进行程序设计，逻辑地址由段基地址和偏移地址组成。编程使用的逻辑地址由处理器映射为线性地址，在输出之前转换为物理地址。

逻辑地址的段基地址部分由 16 位的段寄存器确定。在 IA-32 处理器中，段寄存器仍然是 16 位，但保存的内容是段选择器（Segment Selector）。段选择器是一种特殊的指针，指向对应的段描述符（Descriptor），段描述符包括段基地址，由段基地址就可以指明存储器中的一个段。段描述符是保护方式引入的数据结构，用于“描述”逻辑段的属性。每个段描述符有 3 个字段：段基地址、段长度和该段的访问权字节（说明该段的访问权限，用于特权保护）。

根据存储模型不同，段寄存器的具体内容也有所不同。编写应用程序时，程序员利用汇编程序的命令创建段选择器，操作系统创建具体的段选择器内容。如果编写系统程序，程序员可能需要直接创建段选择器。

平展存储模型下，6 个段寄存器都指向线性地址空间的地址 0 位置，即段基地址等于 0，偏移地址等于线性地址。应用程序通常设置两个重叠的段：一个用于代码，一个用于数据和堆栈。CS 段寄存器指向代码段，其他段寄存器都指向数据和堆栈段。

使用段式存储模型时，段寄存器保存不同的段选择器，指向线性地址空间不同的段。某个时刻，程序最多可以访问 6 个段。CS 指向代码段，SS 指向堆栈段，DS 等其他 4 个段寄存器指向数据段。段式存储管理的段基地址和偏移地址都是 32 位，段基地址加上偏移地址形成线性地址。

实地址和虚拟 8086 方式采用实地址存储模型，只支持 1MB 主存空间。实地址存储模型也进行分段管理，但有两个限制：每个段最大为 64KB，段只能开始于低 4 位地址全为 0 的物理地址处。段寄存器直接保存段基地址的高 16 位，只要将逻辑地址中的段地址左移 4 位，加上偏移地址就可得到 20 位地址。

6.2.3 32 位寻址方式

为了说明指令执行所需的操作数，处理器设计了多种方法指明操作数的位置，这就是寻址方式。操作数可以由指令本身直接提供（立即寻址方式），也可以在处理器内部的通用寄存器中（寄存器寻址方式），很多情况是在存储器中（各种存储器寻址方式）。当然，对外设操作时，操作数来自 I/O 端口。IA-32 处理器指令带有 0~4 个操作数，有些指令不含操作数，多数指令具有 1 或 2 个操作数，个别指令带 3 或 4 个操作数。

IA-32 处理器不仅支持原有的 16 位寻址方式，还增加了灵活的 32 位寻址方式。16 位存储器寻址方式的组成公式为：

$$16 \text{ 位有效地址} = \text{基址寄存器} + \text{变址寄存器} + 8 / 16 \text{ 位位移量}$$

其中基址寄存器只能是 BX 或 BP，变址寄存器只能是 SI 或 DI。

32 位存储器寻址方式的组成公式为：

$$32 \text{ 位有效地址} = \text{基址寄存器} + (\text{变址寄存器} \times \text{比例}) + \text{位移量}$$

其中的 4 个组成部分是：

- ⊙ 基址寄存器——任何 8 个 32 位通用寄存器之一。
- ⊙ 变址寄存器——除 ESP 之外的任何 32 位通用寄存器之一。
- ⊙ 比例——可以是 1、2、4、8（因为操作数的长度可以是 1、2、4、8 字节）。
- ⊙ 位移量——可以是 8、32 位值。

下面用传送指令中的源操作数示例各种 32 位寻址方式，参见表 6-2。

表 6-2 32 位寻址方式举例

寻址方式	指令举例	指令代码（16 位平台）
立即数寻址	mov eax,44332211h	66 B8 11 22 33 44
寄存器寻址	mov eax,ebx	66 8B C3
直接寻址	mov eax,[1234h]	66 A1 34 12 00 00
寄存器间接寻址	mov eax,[ebx]	67 66 8B 03
寄存器相对寻址	mov eax,[ebx+80h]	67 66 8B 83 80 00 00 00
基址变址寻址	mov eax,[ebx+esi]	67 66 8B 04 1E
相对基址变址寻址	mov eax,[ebx+esi+80h]	67 66 8B 84 1E 80 00 00 00
带比例的变址寻址	mov eax,[esi*2]	67 66 8B 04 75 00 00 00 00
基址的带比例的变址寻址	mov eax,[ebx+esi*4]	67 66 8B 04 B3
相对基址的带比例的变址寻址	mov eax,[ebx+esi*8+80h]	67 66 8B 84 F3 80 00 00 00
I/O 端口的直接寻址	in eax,80h	66 E5 80
I/O 端口的寄存器间接寻址	in eax,dx	66 ED

IA-32 处理器的寻址方式覆盖了绝大多数高级语言所需要的数据访问，使高级语言的执行更加有效。使用中需要留心的是，在以 BP、EBP 或 ESP 作为基址寄存器访问存储器数据时，默认的段寄存器是 SS；当 EBP 作为变址寄存器使用时，不影响默认段寄存器的选择。所有其他寻址方式下的存储器数据访问都使用 DS 作为默认段寄存器，包括没有基址寄存器的情况。此外，可以显式地在指令中指定 CS、SS、ES、FS 和 GS 作为访问存储器数据时所引用的段寄存器，以改变默认的段寄存器。

对比 8086 支持的 16 位存储器寻址和 32 位存储器寻址，可以看到：

① 8086 支持的基址寄存器只能是 BX 和 BP，前者用于访问数据段，后者用于访问堆栈段。而 IA-32 中任何一个 32 位通用寄存器都可以用做基址寄存器，EBP 用于寻址堆栈段，其余用于寻址数据段。8086 支持的变址寄存器只能是 SI 和 DI，而 IA-32 可以使用除 ESP 之外任何 32 位通用寄存器。由此可见，IA-32 处理器的 32 位通用寄存器相对 8086 的 16 位通用寄存器更加通用，在编程应用中更加方便。

② 32 位存储器寻址增加了带比例的变址寻址方式，而且这个比例值是 1、2、4 和 8，正好对应 8、16、32 和 64 位数据的 1、2、4 和 8 字节。利用这个寻址方式，指令可以更方便地访问数组。例如下面一个程序片段，由 32 位整数元素组成的数组 ARRAY，如果用寄存器间接寻址“ARRAY[EBX]”循环访问每个元素，初值设置 EBX 为 0，每次 EBX 需要增加 4，EBX 相当于数组元素的地址指针。如果改用带比例的变址寻址“ARRAY[EBX*4]”，每次 EBX 只需要加 1，EBX 就相当于数组元素的编号。

```
array    dd 1234,5678,90
xor ebx,ebx
;寄存器间接寻址
mov eax,[ebx]
add ebx,type array
;带比例的变址寻址
mov eax,[ebx*type array]
inc ebx
```

6.2.4 32 位指令代码

IA-32 处理器的指令系统采用可变长度指令格式，指令编码非常复杂。这一方面是为了兼容 8086 指令，另一方面是为了向编译程序提供更有用的指令。图 6-3 是 IA-32 处理器指令代码的一般格式。它包括几个部分：可选的指令前缀、1~3 字节的主要操作码、可选的寻址方式域（包括 ModR/M 和 SIB 字段）、可选的位移量和可选的立即数。指令前缀和主要操作码字段对应指令的操作码部分，其他字段对应操作数部分。



图 6-3 IA-32 处理器指令的代码格式

1. 指令前缀

指令前缀（Prefix）是指令之前的辅助指令（也称前缀指令），用于扩展指令功能。每个指令之前可以有 0~4 个前缀指令，顺序任意，可以分成 4 组。

第 1 组有 LOCK 前缀指令（指令代码为 F0H），控制处理器总线产生锁定操作。使用 LOCK 前缀后，指令的执行过程中，不允许其他处理器访问共享存储器中的数据，保证了数据的唯一性。第 1 组中还包括仅用于串操作指令的重复前缀指令：REP、REPE / REPZ、REPNE / REPNZ，用于控制串操作指令重复执行。

第 2 组主要是段超越（Segment Override）前缀指令，用于明确指定数据所在段。它们是 CS、DS、SS、ES、FS、GS，对应的指令代码依次是 2EH、3EH、36H、26H、64H、65H。

第 3 组是操作数长度超越（Operand-size Override）前缀，指令代码为 66H。

第 4 组是地址长度超越（Address-size Override）前缀，指令代码为 67H。某条指令单独或同时使用了操作数长度超越前缀和地址长度超越前缀，将改变默认长度。

保护方式下，IA-32 处理器通过段描述符可以为当前运行的代码段选择默认的地址和操作数长度：32 位地址和操作数长度，或者 16 位地址和操作数长度。使用 32 位地址长度，偏移地址是 FFFFFFFFH ($2^{32}-1$)，逻辑地址由一个 16 位段选择器和一个 32 位偏移地址组成。使用 16 位地址长度，最大偏移地址是 FFFFH ($2^{16}-1$)，逻辑地址由一个 16 位段选择器和一个 16 位偏移地址组成。32 位操作数长度确定操作数可以是 8 位或者 32 位；16 位操作数长度确定操作数可以是 8 位或者 16 位。例如，当前段默认是 32 位操作数长度和地址长度（可称为 32 位段），指令如果使用了操作数长度超越前缀 66H，则指令的操作数实际上是 16 位；如果采用了 16 位地址的寻址方式，则指令的存储器寻址在 16 位段，如表 6-3 右列所示。

表 6-3 操作数长度超越和地址长度超越的作用

16 位代码段中的指令代码	汇编指令	地址和操作数长度	32 位代码段中的指令代码
8B C3	mov ax, bx	16 位操作数	66 8B C3
66 8B C3	mov eax, ebx	32 位操作数	8B C3
8B 07	mov ax, [bx]	16 位操作数、16 位寻址	67 66 8B 07
67 8B 03	mov ax, [ebx]	16 位操作数、32 位寻址	66 8B 03
66 8B 07	mov eax, [bx]	32 位操作数、16 位寻址	67 8B 07
67 66 8B 03	mov eax, [ebx]	32 位操作数、32 位寻址	8B 03

实地址方式、虚拟 8086 方式、系统管理方式默认采用 16 位地址和操作数长度（可称为 16 位段）。但指令也可以使用两个长度超越前缀，采用 32 位操作数和地址长度，如表 6-3 左列。不过，此时采用 32 位地址长度所访问的线性地址最大仍然是 000FFFFH ($2^{20}-1$)。

2. 操作码和操作数

指令执行的操作（如加、减、传送等）编码为操作码部分。主要操作码是 1~3 字节，有些还用到 ModR/M 中的 3 位。

IA-32 处理器设计有多种存取操作数的方法，所以操作数的编码（地址码）比较复杂。寻址方式的 ModR/M 和 SIB 字段提供操作数地址信息。例如，它们指明操作数是在指令代码中，还是在寄存器或存储器中。如果操作数在指令代码中，立即数字段就是所需要的操作数；如果操作数在存储器中，则需要进一步指明采用何种方式访问存储器，有时还需要相对起始地址的位移量。

例如，将 32 位寄存器 EBX 的数据传送到 EAX 寄存器的指令，可以书写为：

```
mov eax,ebx
```

这个指令的机器代码在 32 位平台是：8B C3（十六进制，下同）。其中，“8B”是操作码，“C3”表达操作数。如果使用 16 位操作数形式（使用 16 位寄存器 BX 和 AX），即指令“MOV AX,BX”，那么它的机器代码是：66 8B C3。这里的“66”就是操作数长度超越前缀。如果是 16 位平台，则指令“MOV AX,BX”的机器代码是：8B C3；而指令“MOV EAX,EBX”的机器代码是：66 8B C3。由此可以体会操作数长度超越前缀的作用：改变默认的操作数长度时需要使用操作数长度超越前缀。

再如，将由 EBX 指明偏移地址的存储器内的数据传送 EAX，可以书写为：

```
mov eax,[ebx]
```

这个指令的机器代码在 32 位平台是：8B 03。其中，“03”字节由 ModR/M 字段生成。在 16 位平台，该指令的机器代码是：66 67 8B 03。其中操作数长度超越前缀“66”，表示在默认使用 16 位操作数的平台中使用 32 位操作数。同时注意，其中的“67”就是地址长度超越前缀，表示在默认使用 16 位地址的平台中使用 32 位地址的访问形式。

如果数据不是在默认的 DS 数据段，需要使用段超越前缀显式说明。例如：

```
mov eax,es:[ebx]
```

该指令用“ES:”表达数据在 ES 段，它的机器代码（32 位平台）是：26 8B 03。这里的“26”就是 ES 段超越前缀。

IA-32 支持复杂的数据寻址方法，例如：

```
mov eax,[ebx+esi*4+80h]
```

这个指令中，数据来自主存数据段，偏移地址由 ESI 内容乘以 4 加 EBX、再加位移量 80H 组成。它的机器代码（32 位平台）是：8B 84 B3 80 00 00 00。其中“84”由 ModR/M 字段生成，“B3”由 SIB 字段生成，后面 4 字节表达位移量 00000080H。

6.3 32 位整数指令系统

在 IA-32 整数指令集中，8086 的 16 位指令仍是最基本、最常用的指令。IA-32 处理器只是一方面扩展原来指令支持 32 位操作数和 32 位寻址方式（本书称它们为 32 位扩展指令，也包括 80186 和 80286 新增的 16 位指令），另一方面新增了部分有特色的指令。

类似 16 位指令学习，增加几个特定符号来表达 32 位操作数：

- ⊙ r32—— 一个 32 位通用寄存器，EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP。
- ⊙ m32—— 一个 32 位存储器操作数单元。
- ⊙ i32—— 一个 32 位立即数。

这样，原来使用的 reg、mem 和 imm 符号也分别包括了上述 32 位操作数，seg 包括了 FS/GS 段寄存器。

6.3.1 32 位扩展指令

在 16 位最基本、最常用指令基础上，32 位指令系统从两方面向 32 位扩展：一是所有指令都可扩展支持 32 位操作数，包括 32 位的立即数；二是所有涉及存储器寻址的指令都可以使用 32 位的寻址方式。例如：

```
mov ax,bx           ;16 位操作数
mov eax,ebx         ;32 位操作数
mov ax,[ebx]        ;16 位操作数，32 位寻址方式
mov eax,[ebx]       ;32 位操作数，32 位寻址方式
```

另外，有些指令扩大了工作范围，或指令功能实现了向 32 位的自然增强。

下面按功能分类介绍这些 32 位扩展指令，重点说明这些指令对 32 位操作数的支持，以及在 16 位段与 32 位段中的异同。

1. 数据传送类指令

数据传送类指令实现在寄存器、存储单元或 I/O 端口之间的数据传输。

(1) 通用传送

传送指令 MOV 的格式、功能和使用注意事项都与 16 位 8086 的 MOV 指令相同，现在除可以进行 8 位或 16 位数值传送外，还可以进行 32 位数值的传送。再如：

```
mov dword ptr [di],eax ;DWORD PTR 伪指令指明传送 32 位双字数值
mov al,[ebp+ebx]       ;默认采用 SS 段寄存器（因为 EBP 用做基址寄存器）
mov ah,fs:[5678h]      ;显式指定采用 FS 段寄存器
```

交换指令 XCHG 可以实现 8 位、16 位和 32 位数据的交换，例如：

```
xchg esi,edi
xchg [ebx],cx
```

(2) 堆栈操作

进栈 PUSH 和出栈 POP 指令除可以对 16 位数据进行操作外，现在还可以将 32 位寄存器和存储单元内容压入和弹出堆栈，当然，堆栈指针（E）SP 应该减或加 4。例如：

```
push eax
pop dword ptr [bx]
```

从 80186 开始，PUSH 指令可以将立即数压入堆栈：

```
push i8/i16/i32      ;把 16 位或 32 位立即数 i16 / i32 压入堆栈
                    ;若是 8 位立即数 i8，则经符号扩展成 16 位后再压入堆栈
```

在利用堆栈传递参数时，把立即数压入堆栈可以方便地实现将常量作为参数传递给子程序。例如：

```
push word ptr 1234h   ;压入 16 位立即数
push dword ptr 87654321h ;压入 32 位立即数
call helloabc         ;调用子程序
add esp,6             ;平衡堆栈
```

为了更方便地进行寄存器保护与恢复，80186 引入了 16 位通用寄存器进栈 PUSH 和出栈 POP 指令。

pusha	;顺序将 AX/CX/DX/BX/SP/BP/SI/DI 压入堆栈
popa	;顺序从堆栈弹出 DI/SI/BP/SP/BX/DX/CX/AX (与 PUSH 相反)
	;应进入 SP 的值被舍弃，SP 通过增加 16 来恢复

同样，80386 引入了 32 位通用寄存器进栈 PUSHAD 和出栈 POPAD 指令。

pushad	;顺序将 EAX/ECX/EDX/EBX/ESP/EBP/ESI/EDI 压入堆栈
popad	;顺序从堆栈弹出 EDI/ESI/EBP/ESP/EBX/EDX/ECX/EAX
	;应进入 ESP 的值被舍弃，ESP 通过增加 32 来恢复

由于 IA-32 处理器增加了 FS 和 GS 段寄存器，所以也可以对它们进行堆栈操作。

此外，有一点需要注意：当用 PUSH 指令把堆栈指针 (E) SP 压入堆栈时，80286 以后的处理器是将进栈前的 (E) SP 值进栈；而 8086/8088 则是将 SP 减 2 后的值进栈。

(3) 其他标志传送

80386 引入 PUSHFD 和 POPFD 指令用于通过堆栈传送 32 位标志寄存器内容。

装入有效地址 LEA 指令可以取 16 或 32 位有效地址，在 16 位段和 32 位段有一定差别。

lea r16,mem	;16 位段中，计算存储器单元的 16 位有效地址送 r16
	;32 位段中，计算 32 位有效地址，但取其低 16 位送 r16
lea r32,mem	;16 位段中，计算存储器单元的 16 位有效地址，经零位扩展后送 r32
	;32 位段中，计算存储器单元的 32 位有效地址送 r32

例如：

mov ebx,12345678h	
lea dx,[ebx+4321h]	;执行后 DX=9999H
lea esi,[ebx+1111h]	;32 位段中，ESI=12346789H
	;16 位段中，ESI=00006789H，因仅低 16 位有效，高 16 位为 0
lea edi,[bx+2222h]	;EDI=0000789AH

换码指令 XLAT 如果在 16 位段中，则该指令与 8086 相同；如果是在 32 位段中，则该指令将采用 EBX 存放基值。

对 I/O 端口的操作除可以利用 AL 和 AX 输入输出一字节或字外，现在还支持通过 EAX 的双字数据操作。例如：

```
out 20h,eax
in  eax,dx
```

2. 算术运算类指令

算术运算指令完成加、减、乘、除及求补和比较等操作。

8086 的加法和减法指令有 8 种，IA-32 处理器将操作数扩展到 32 位。

例如，32 位的乘法和除法指令如下：

mul imul r32/m32	;双数值乘法：EDX:EAX←EAX×r32/m32
div idiv r32/m32	;双数值除法：EAX←EDX:EAX÷r32/m32 的商
	;EDX←EDX:EAX÷r32/m32 的余数

从 80186 开始，有符号数乘法又提供了新形式：

imul r16,r16/m16,i8/i16	;r16←r16×r16/m16/i8/i16
imul r16,r16/m16,i8/i16	;r16←r16/m16×i8/i16
imul r32,r32/m32,i8/i32	;r32←r32×r32/m32/i8/i32
imul r32,r32/m32,i8/i32	;r32←r32/m32×i8/i32

这些新增乘法形式的目的和源操作数，包括立即数的长度都相同（对于 8 位立即数 i8 的情况，它的高位要进行符号扩展），因此乘积有可能溢出。如果乘积溢出，那么高位部分被丢掉，并置 CF=OF=1；如果没有溢出，则 CF=OF=0。例如：

```
imul eax,10
imul ebx,ecx
imul ax,bx,-2
imul eax,dword ptr [esi+8],5
```

后一种形式采用了 3 个操作数，而前一种形式实际上可以认为是后一种形式的特殊情况。例如：

```
imul ax,7 ;等同于：imul ax,ax,7
```

由于存放积的目的操作数长度与乘数的长度相同，而有符号数和无符号数的乘积的低位部分相同，所以这种新形式的乘法指令对有符号数和无符号数的处理是相同的。

80386 一方面在原来 CBW 和 CWD 指令的基础上扩展了两条符号扩展指令：

```
cwde ;把 AX 符号扩展为 EAX，该指令是 CBW 的扩展
cdq ;把 EAX 符号扩展为 EDX.EAX，该指令是 CWD 的扩展
```

同时，80386 扩展了更加方便使用的零位扩展 MOVZX 和符号扩展 MOVSX 指令：

```
movsx r16,r8/m8 ;把 r8/m8 符号扩展并传送至 r16
movsx r32,r8/m8/r16/m16 ;把 r8/m8/r16/m16 符号扩展并传送至 r32
movzx r16,r8/m8 ;把 r8/m8 零位扩展并传送至 r16
movzx r32,r8/m8/r16/m16 ;把 r8/m8/r16/m16 零位扩展并传送至 r32
```

可以看到它们的功能更强。例如：

```
mov bl,92h
movsx ax,bl ;AX=FF92H
movsx esi,bl ;ESI=FFFFFF92H
movzx edi,ax ;EDI=0000FF92H
```

3. 位操作类指令

这类指令实现对操作数的按二进制位的操作。

逻辑运算指令的操作数现在扩展到 32 位。移位及循环移位指令现在也支持 32 位操作数，而且从 80186 开始，它们还可以用一个立即数指定大于 1 的移位次数。例如：

```
shl al,4
sar eax,12
rcr word ptr [si],3
```

4. 串操作类指令

串操作指令方便了对数组类型数据的操作。这类指令除可以进行字节操作和字操作外，还可以实现双字操作，用后缀字母 D 表示，分别为 MOVSD、LODSD、STOSD、CMPSD 和 SCASD。在进行双字操作时，(E) SI 和 (E) DI 进行加 4（标志 DF=0）或减 4（标志 DF=1）。在 16 位段中，用 SI 和 DI 指示存储器偏移地址；而在 32 位段中，用 ESI 和 EDI 指示偏移地址。

串操作指令可以使用重复前缀 REP、REPZ/REPE 和 REPNZ/REPNE。在 16 位段中用 CX，而在 32 位段中用 ECX 作为重复次数的计数器。

8086 只能通过 IN 和 OUT 指令对 I/O 端口进行数据传送。从 80186 开始，对 I/O 端口的

操作也可以采用串操作，配合重复前缀就能够实现用一条指令连续进行输入或输出，极大地提高了 I/O 操作能力。从 80386 开始，还可以进行 32 位数据的输入和输出。

ins	;I/O 串输入: ES:[(E)DI]←I/O 端口[DX], (E)DI←(E)DI±1/2/4
outs	;I/O 串输出: I/O 端口[DX]←DS:[(E)SI], (E)SI←(E)SI±1/2/4

5. 控制转移类指令

控制转移指令使程序改变了执行顺序，从一处跳到了另一处。

无条件转移指令 **JMP** 仍分为段内相对、段内间接、段间直接和段间间接转移 4 类。IA-32 处理器由于具有 32 位段，所以，转移的目标偏移地址可以扩展为 32 位，段间转移目的地址也可以采用 48 位全指针形式（16 位段寄存器：32 位偏移地址）。

JCC 指令的条件没有变化，但在 IA-32 处理器中，允许采用多字节来表示转移目的偏移与当前偏移之间的差，所以转移范围可以超出原来的一128~+127，达到 32 位的全偏移量（在 4GB 线性地址范围内）。这一点增强了原来这些指令的功能，使得程序员不必担心条件转移是否超出了范围。

在 16 位段中，循环指令保持原功能，但在 32 位段中，使用 **ECX** 作为计数器，即从 **CX** 扩展到了 **ECX**。IA-32 处理器在 16 位段使用 **JCXZ** 指令（**CX**=0 跳转），而在 32 位段使用新增的 **JECXZ** 指令（**ECX**=0 跳转）。另外，这些循环指令的转移范围仍是一128~+127。

在 32 位段中，过程调用 **CALL** 指令的转移目的地址的偏移为 32 位；段间转移目的地址采用 48 位全指针形式，并且在把返回地址的 **CS** 压入堆栈时扩展成高 16 位为 0 的双字，这样将压入堆栈 2 个双字共 8 个字节。同样，在 32 位段中，过程返回 **RET** 指令要从堆栈弹出双字作为返回地址的偏移；对段间返回，要从堆栈弹出包含 48 位全指针的 2 个双字。

为了更好地支持高级语言，方便编写编译程序，从 80186 开始引入了 3 条新指令，它们是：**ENTER**、**LEAVE**、**BOUND**。

另外，80286 虽然也是一个 16 位处理器，但它引入了保护方式，极大地提高了它的性能。相应地，80286 就设计有一些用于保护方式的指令，同样适合于 IA-32 处理器。这些指令很多都是所谓的特权指令，通常只有系统核心程序能够使用它们。

6.3.2 32 位新增指令

32 位指令系统在原来 8086、80186 和 80286 指令系统基础上，新增了很有特色的整数指令，本节选择有特色的部分指令进行简介。

1. 80386 新增指令

80386 处理器的执行单元中新增了一个“桶型”移位器（实现快速位移操作的硬件电路），可以实现快速移位操作，80386 新增的指令主要是有关位操作的。例如，双精度左移 **SHLD**、双精度右移 **SHRD**、前向扫描 **BSF**、后向扫描 **BSR**、位测试指令等。另外，80386 还增加了条件设置指令，以及对控制、调试和测试寄存器的传送指令等。

（1）位测试指令

80386 新增的 4 条位测试指令比较常用，如下所示：

bt dest,src	;把目的操作数 dest 中由源操作数 src 指定的位送 CF 标志
btc dest,src	;把 dest 中由 src 指定的位送 CF 标志，然后对那一位求反
btr dest,src	;把 dest 中由 src 指定的位送 CF 标志，然后对那一位复位

bts dest,src ;把 dest 中由 src 指定的位送 CF 标志, 然后对那一位置位

这 4 种指令中, 目的操作数 DEST 只能是 16、32 位通用寄存器或存储单元, 用于指定要测试的数据; 源操作数 SRC 必须是 8 位立即数或者是与目的操作数等长的 16、32 位通用寄存器, 用于指定要测试的位。

如果目的操作数是寄存器, 则源操作数除以 16 或 32 的余数就是要测试的位, 它在 0~15 (31) 之间。例如:

```
mov eax,12345678h ;EAX=12345678H
bt eax,5           ;EAX=12345678H, CF←1=EAX 的 D5 位
btc eax,10          ;EAX=12345278H, CF←1=EAX 的 D10 位
btr eax,20          ;EAX=12245278H, CF←1=EAX 的 D20 位
bts eax,34          ;EAX=1224527CH, CF←0=EAX 的 D2 位
```

如果目的操作数是存储单元, 则该单元的最低位为 0; 从这个最低位向地址高端每位依次增量, 向地址低端每位依次减量, 这部分存储器数据作为一个 2G-1~2G 长的位串。此时, 有符号源操作数就指示要测试的位。例如:

```
bitsa    dw 1234h,5678h,9abcdh
bitsb    dd 12345678h
...
bt bitsa,4      ;[BITSA]=1234H, CF←1
mov cx,22
btc bitsa,cx    ;[BITSA+2]=5638H, CF←1
movzx eax,cx
bts bitsb,eax   ;[BITSB]=12745678H, CF←0
```

(2) 条件设置指令

在采用流水线技术的现代处理器中, 条件转移指令是影响性能的重要因素之一。所以, 最好能够不使用条件转移指令而实现分支程序结构。为此, 80386 新增了条件设置指令, 用于减少程序中的条件转移指令。

setcc r8/m8 ;若条件 CC 成立, 则 r8/m8 为 1; 否则, 为 0

本指令根据处理器定义的 16 种条件 CC, 设置所指定的字节量。这 16 种条件与条件转移指令 JCC 中的条件是一样的。例如:

setcc al ;如果 CF=1, 则 AL=1; 否则, AL=0

下面用 JCC 指令实现一个典型的双分支程序段:

```
cmp eax,i32
jge L00
mov ebx,C1 ;EAX<i32: EBX←C1
jmp L01
L00: mov ebx,C2 ;EAX≥i32: EBX←C2
L01: ...
```

如果使用 SETCC 指令完成同一个功能, 就可以消除这个分支:

```
xor edx,edx
cmp eax,i32
setge bl ;EAX≥i32: BL←1 ; EAX<i32: BL←0
dec ebx ;EAX≥i32: EBX←0 ; EAX<i32: EBX←FFFFFFFFH
and ebx,(C1-C2) ;EAX≥i32: EBX←0 ; EAX<i32: EBX←(C1-C2)
add ebx,C2 ;EAX≥i32: EBX←C2 ; EAX<i32: EBX←C1
```

2. 80486 新增指令

80486 处理器的指令系统在 80386 指令集的基础上增加了 6 条新指令，新增的指令主要用于对多处理器系统和片上高速缓冲存储器的支持。下面介绍 3 条常用指令。

(1) 字节交换指令 BSWAP

bswap r32 ;将 32 位通用寄存器值的第 1 和 4 字节、第 2 和 3 字节互换

例如：

mov eax,00112233h ;EAX=00112233h

bswap eax ;EAX=33221100h

Intel 80x86 系列处理器采用低字节数据存于低地址的小端方式存储多字节数据，而许多精简指令集 RISC 处理器则采用低字节数据存于高地址的大端方式。字节交换指令 BSWAP 可以方便地进行这两种存储格式的相互转换。

(2) 交换加指令 XADD

xadd r8/m8,r8 ;r8/m8 \leftrightarrow r8, r8/m8 \leftarrow r8+r8/m8

xadd r16/m16,r16 ;r16/m16 \leftrightarrow r16, r16/m16 \leftarrow r16+r16/m16

xadd r32/m32,r32 ;r32/m32 \leftrightarrow r32, r32/m32 \leftarrow r32+r32/m32

交换加指令首先将目的和源操作数互换，然后将两者之和送到目的操作数，按照加法指令影响标志 OF、SF、ZF、AF、PF、CF。例如：

mov bl,12h

mov dl,02h

xadd bl,dl ;BL=14h, DL=12h

(3) 比较交换指令 CMPXCHG

cmpxchg r8/m8,r8 ;AL \leftarrow r8/m8; 相等: ZF=1, r8/m8 \leftarrow r8

;不等: ZF=0, AL \leftarrow r8/m8

cmpxchg r16/m16,r16 ;AX \leftarrow r16/m16; 相等: ZF=1, r16/m16 \leftarrow r16

;不等: ZF=0, AX \leftarrow r16/m16

cmpxchg r32/m32,r32 ;EAX \leftarrow r32/m32, 相等: ZF=1, r32/m32 \leftarrow r32

;不等: ZF=0, EAX \leftarrow r32/m32

比较交换指令比较累加器 AL、AX 或 EAX 和目的操作数，如果相等，把源操作数送给目的操作数，并置位 ZF；如果不相等，则把目的操作数送给累加器，并复位 ZF。该指令按照比较指令影响标志 OF、SF、ZF、AF、PF、CF。例如：

mov al,12h

mov bl,12h

mov dl,02h

cmpxchg bl,dl ;AL=12H, BL \leftarrow DL=02H, ZF=1

cmpxchg bl,dl ;AL \leftarrow BL=02H, DL=02H, ZF=0

3. Pentium 新增指令

Pentium 指令系统是 80486 指令集的超集，新增了几条非常实用的指令。

例如，处理器识别指令 CPUID 是一个很有特色的指令：

cpuid ;返回处理器的有关特征信息

随着 80x86 系列处理器不断升级换代，新的处理器具有越来越强的功能和指令。虽然，原来在老型号处理器上运行的程序，在新型号处理器上仍然可以运行，但是新开发的程序将会使用新增指令，程序也只有利用处理器提供的新特征，才能充分发挥处理器的能力，达到

最佳的运行效果。所以，一个优秀的程序应该能够根据不同的处理器采用不同的方法，实现相同的功能。这里，首先必须解决的问题就是识别出不同的处理器型号。

对前几代 80x86 处理器的识别，常通过判断标志寄存器中某些特定标志来实现。从 Pentium 开始，处理器提供了 CPU 识别指令，后期生产的某些 80486 芯片也支持该指令。通过确认 CPU 识别标志 ID（EFLAGS 寄存器的 D21 位）位改变，就可以判断出该处理器支持 CPUID 指令。执行 CPUID 指令前，必须给 EAX 赋入口参数。EAX 值不同，CPUID 指令返回的信息不同。

再如，读时间标记计数器指令 RDTSC 指令可以精确地检测程序执行时间：

```
rdtsc                ;EDX:EAX←64 位时间标记计数器值
```

Pentium 含有一个 64 位的时间标记计数器（Time-Stamp Counter）。该计数器每个时钟周期递增（加 1）；在上电和复位后，该计数器清 0。在欲测试程序执行前，利用 RDTSC 指令获得一个时钟数值，程序执行后再次得到一个时钟数值，后者减去前者就是欲测试程序执行所花费的时钟周期数，进而结合处理器时钟频率就可以计算出程序执行时间。

4. Pentium Pro 新增指令

对比 Pentium 指令集，Pentium Pro 的指令系统新增了 3 条实用的指令。

例如，80386 引入条件设置指令用于减少条件转移指令的使用，Pentium Pro 再进一步新增条件传送指令 CMOVCC，其中条件 CC 同 JCC 和 SETCC 指令：

```
cmovcc r16,r16/m16    ;若条件 CC 成立，则 r16←r16/m16；否则，不传送  
cmovcc r32,r32/m32    ;若条件 CC 成立，则 r32←r32/m32；否则，不传送
```

该指令首先判断条件是否满足。如果条件成立，则发生传送：源操作数传送到目的操作数；如果条件不成立，则不进行传送，好像没有执行该指令一样。

下面是一个典型的单分支程序段：

```
test ecx,ecx          ;判断 ECX 是否等于 0  
jne next  
mov eax,ebx           ;ECX=0，则 EAX←EBX  
next: ...              ;ECX≠0
```

如果采用条件传送指令，则可以消除分支结构，优化为：

```
test ecx,ecx          ;CMOVEQ 有条件地将 EBX 传送到 EAX  
cmoveq eax,ebx        ;代替 JNE 与 MOV 指令，从而消除分支
```

6.4 DOS 平台的 32 位指令编程

在 DOS 平台，利用 32 位指令进行汇编语言程序设计的方法同前几章介绍的 16 位指令程序设计基本相同。但编写完整的汇编语言源程序，还需要注意一些地方。

1. 指定汇编程序识别新指令

MASM 在默认情况下只汇编 16 位 8086 处理器的指令（前 5 章介绍的处理器指令）；如果需要使用 80186 及以后处理器新增的指令，必须使用处理器选择伪指令，如表 6-4 所示。

例如，当源程序中使用了 80286 新增的指令时，应该在程序中加上 .286 或 .286P 伪指令；若想利用 32 位寄存器完成 32 位操作，则必须加上 .386 及以上处理器的选择伪指令。

表 6-4 处理器选择伪指令

伪 指 令	功 能	伪 指 令	功 能
.8086	仅接受 8086 指令（默认状态）	.586	接受除特权指令外的 Pentium 指令
.186	接受 80186 指令	.586P	接受全部 Pentium 指令
.286	接受除特权指令外的 80286 指令	.686	接受除特权指令外的 Pentium Pro 指令
.286P	接受全部 80286 指令，包括特权指令	.686P	接受全部 Pentium Pro 指令
.386	接受除特权指令外的 80386 指令	.MMX	接受 MMX 指令
.386P	接受全部 80386 指令，包括特权指令	.K3D	接受 AMD 处理器的 3D 指令
.486	接受除特权指令外的 80486 指令，包括浮点指令	.XMM	接受 SSE, SSE2 和 SSE3 指令
.486P	接受全部 80486 指令，包括特权指令和浮点指令	注：.586/.586P 由 MASM 6.11 引入； .686/.686P/.MMX 由 MASM 6.12 引入； .K3D 由 MASM 6.13 引入； .XMM 由 MASM 6.14 引入，MASM 6.15 支持 SSE2 指令，MASM 8.0 支持 SSE3 指令。	
.8087	接受 8087 数学协处理器指令		
.287	接受 80287 数学协处理器指令		
.387	接受 80387 数学协处理器指令		
.No87	取消使用协处理器指令		

另外，书写 .386 及以后的处理器选择伪指令需要留心其位置，如果书写在存储模型 MODEL 指令之后，该程序将默认采用实方式和虚拟方式使用的 16 位段（默认是 16 位地址和操作数长度）；如果书写在存储模型 MODEL 指令之前，该程序将默认采用保护方式使用的 32 位段（默认是 32 位地址和操作数长度）。

2. 留意 16 位段和 32 位段的差别

针对 32 位 IA-32 处理器，编写 DOS 环境（实方式和虚拟 8086 方式）的可执行程序时，尽管可以利用处理器的 32 位寄存器、32 位寻址方式，主要处理 32 位数据、进行 32 位操作和执行 32 位新增指令，但程序的逻辑段仍然必须是 16 位段，即最大 64KB 的物理段。只有进入了保护方式，才可以使用 32 位段。但在 DOS 环境中，可以编辑、汇编 32 位段程序和开发只能在 32 位段运行的程序。

由于 16 位段和 32 位段的属性不同，有些指令在 16 位段和 32 位段的操作会有差别。例如，串操作指令在 16 位段采用 SI 和 DI 指示地址、CX 表达个数；而在 32 位段采用 ESI 和 EDI 指示地址、ECX 表达个数。循环指令一样，在 16 位段采用 CX 记数；在 32 位段采用 ECX 记数。但是，由于在 32 位通用寄存器中的高 16 位无法单独直接利用，所以即使在 16 位段采用 32 位寄存器也无妨，只是注意指令只利用了低 16 位，高 16 位没有使用，最好为 0。这样处理的另一个好处是，当将这个 16 位段的程序段用于 32 位段时通常无须修改。

另外，由于处理 32 位数据，变量定义等伪指令要经常采用双字 DWORD 属性。

【例 6-1】 处理器生产商识别程序。

知道 Intel 或 AMD 公司为其处理器内置的识别字符串是什么吗？从 Pentium 开始，Intel 处理器提供了处理器识别指令 CPUID；后期生产的某些 80486 芯片也支持该指令。

当 EAX=0 时执行 CPUID 指令，将通过 EBX、EDX 和 ECX 返回生产厂商的标识串。Intel 公司的处理器是“GenuineIntel”，这 3 个寄存器依次存放 Genu、ineI、ntel 的 ASCII 码，如图 6-4 所示。利用这个厂商标识串，就能确认是 Intel 公司的 IA-32 处理器。当 EAX=1 或 2 等值时执行 CPUID 指令，将进一步返回处理器更详细的识别信息。

```
.model small
.686           ;采用 32 位指令
.stack
```

```

.data
buffer db 'The processor vendor ID is ',12 dup(0),'$'
bufsize = sizeof buffer
.code
.startup
mov eax,0
cpuid          ;执行处理器识别指令
mov dword ptr buffer+bufsize-13,ebx
mov dword ptr buffer+bufsize-9,edx
mov dword ptr buffer+bufsize-5,ecx
mov dx,offset buffer ;显示信息
mov ah,9
int 21h
.exit
end

```

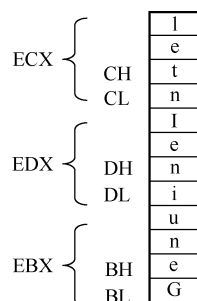


图 6-4 处理器标识串的存放格式

由于使用了 Pentium 开始支持的 CPOUID 指令，源程序中需要使用“.586”或者“.686”处理器选择伪指令。否则，MASM 汇编时会提示错误信息：指令或寄存器不被当前 CPU 模式支持。对比前 5 章的源程序，这个支持 32 位指令的汇编语言源程序最主要就是使用处理器选择伪指令，并且在存储模型 MODEL 之后。

缓冲区 BUFFER 预留了 12 字节空间用于存放标识串，最后一个字符“\$”是 9 号 DOS 功能使用的字符串结尾标志。

AMD 公司的处理器也支持 CPOUID 指令，但返回不同的标识串。在 AMD 公司的处理器上执行本示例程序，看其内置字符串是什么？

【例 6-2】 64 位数据移位程序。

16 位 8086 指令能够直接对 16 位数据进行移位操作，超过 16（如 32 位）需要分段移位（参见例 3-9）。32 位 IA-32 指令则可以直接对 32 位数据进行移位，不过对超过 32 位的数据也需要进行分段移位，例如实现一个 64 位数据算术左移 5 位。

```

.model small
.686          ;采用 32 位指令
.stack
.data
qvar         dq 1234567887654321h ;定义一个 64 位数据变量
.code
.startup
mov eax,dword ptr qvar ;使用双字类型才能实现类型匹配
mov edx,dword ptr qvar[4] ;使用 EDX.EAX 寄存器对保存 64 位数据
mov ecx,5          ;32 位指令使用 ECX 作为循环计数器
again:
shl eax,1
rcl edx,1
loop again
mov dword ptr qvar,eax
mov dword ptr qvar[4],edx
.exit
end

```

本例程序采用的算法类似例 3-9，利用一对 32 位寄存器 EDX 和 EAX 暂存 64 位数据进

行移位。也可以直接对 64 位变量的高低 32 位进行移位操作，如下所示（不过每次移位都要访问主存单元，可能不及通过寄存器移位的性能，尤其是进行多位移动时）：

```
        mov ecx,5           ;32 位指令使用 ECX 作为循环计数器
again:   shl dword ptr qvar,1
        rcl dword ptr qvar[4],1
        loop again
```

【例 6-3】 计算自然数求和的最大值程序。

16 位编码能够表达的无符号整数范围是 0~65535，而 32 位编码则可以表达的范围大大扩展，达到 0~4294967295 ($=2^{32}-1$)。当处理器指令升级为 32 位后，自然能够方便地处理更大范围的数值。

第 4 章例 4-1 实现自然数求和，即 $1+2+3+\cdots+N$ 。不过，由于只是用 16 位寄存器，所以 N 值不能超过 65535。如果使用 32 位寄存器则 N 值可以大大提高，不过 N 最大是多少，和值就要超过 32 位能够表达的范围（4294967295）呢？本例程序求出最大的 N 值，以及对应的自然数和值（当然，如果采用 64 位表达求和值，则 N 值可以达到 32 位所能表达的范围，只是编程就更加复杂了）。

```
        .model small
        .686
        .stack
        .data
maxNum   dd ?           ;保存最大 N 值
maxSum   dd ?           ;保存最大和值
        .code
        .startup
xor ebx,ebx           ;EBX 用于保存和值，初值为 0
xor ecx,ecx           ;ECX 用于保存 N 值，初值为 0
again:   inc ecx        ;从 1 开始
        add ebx,ecx     ;求和
        jnc again      ;没有进位即没有超出范围，继续求和
        sub ebx,ecx     ;有进位，说明超出范围，减去超出部分就是最大和值
        mov maxSum,ebx  ;保存最大和值
        dec ecx        ;减 1 为最大 N 值（因为不减 1 的值使得和值超出了范围）
        mov maxNum,ecx ;保存最大 N 值
        .exit
        end
```

本例程序所采用的方法与例 4-17 输出斐波那契数列相同，出现进位就是超出范围。但是，本例题只是将最大 N 值及对应自然数和值保存在变量中。如果要知道这两个值究竟是多少，需要在支持 32 位指令的调试程序（如 CodeView）中查看，或者需要用户编写一个以十进制形式显示 32 位二进制数编码的子程序（作为习题 6.13，结果：maxNum=92681，maxSum=4294930221）。

由于 Intel 公司在设计 32 位处理器时遵循兼容性原则，所以通过前面指令介绍和源程序学习，可以看到使用 IA-32 处理器的 32 位指令进行汇编语言编程，相对使用 8086 处理器的 16 指令编程并没有本质上的区别。原来使用 16 位寄存器、数据或变量，现在都可以使用 32 位寄存器、数据或变量，自然也就成为 32 位指令。当然，还可以使用新增的 32 位指令。

习 题 6

6.1 简答题

- (1) 对比 8086 的标志寄存器, 32 位标志寄存器 EFLAGS 中的状态标志有变化吗?
- (2) 为什么 32 位存储器寻址设计支持 1、2、4 和 8 的比例?
- (3) 代码为 66H 的超越前缀的作用是什么?
- (4) 代码为 67H 的超越前缀的作用是什么?
- (5) 指令“ADD ECX, BX”有错吗?
- (6) 指令“PUSH EAX”执行后, ESP 发生什么变化?
- (7) 指令 JCC 的转移范围在 IA-32 和 8086 处理器上有什么区别?
- (8) CPUID 指令返回识别字符串的首个字符“G”在哪个寄存器中?
- (9) 如何让 MASM 汇编程序识别 80386 指令?
- (10) LOOP 指令在 16 位段和 32 位段有什么不同?

6.2 判断题

- (1) IA-32 处理器是指 80386 和 80486 处理器, 不包括 Pentium 系列处理器。
- (2) 在 IA-32 处理器中, 寄存器 DX 是 EDX 的低 16 位部分。
- (3) IA-32 处理器在实地址方式下, 不能使用 32 位寄存器。
- (4) “MOV AX, [EBX+ECX]”是一条正确的 80386 指令。
- (5) 指令“SAR EAX, 4”在 8086 中是非法的, 但在 IA-32 处理器中则是合法的。
- (6) IA-32 指令“MOVSX AX, AL”与 8086 指令 CBW 功能不同。
- (7) 新增 32 位指令 SETcc 和 CMOVcc 中条件 cc 与原来 JCC 指令中的 CC 含义相同。
- (8) LOOP 指令跳转范围在 IA-32 处理器上可以超过短转移范围。
- (9) 使用 32 位寄存器, 就不能使用 16 位的 DOS 功能调用。
- (10) 在 16 位 DOS 平台可以开发运行于 32 位 Windows 平台的汇编语言程序。

6.3 填空题

- (1) Intel_____处理器开始将 8086 指令系统升级为 32 位指令系统, 而_____处理器开始内部集成浮点处理单元、开始支持浮点操作指令。
 - (2) IA-32 处理器有_____个段寄存器, 它们都是_____位的。
 - (3) IA-32 处理器复位后, 首先进入是_____工作方式。该工作方式分段最大不超过_____。
 - (4) 已知 ESI=04000H, EBX=20H, 指令“MOV EAX, [ESI+EBX*2+8]”中访问的有效地址是_____。
 - (5) 用 EBX 做基地址指令, 默认采用_____段寄存器指向的数据段; 如果采用 EBP 或 ESP 作为基地址指针, 默认使用_____段寄存器指向堆栈段。
 - (6) 已知 AL=85H, 则零位扩展指令“MOVZX BX, AL”执行后, BX=_____。
- 和指令
- (7) 已知 AL=85H, 则符号扩展指令“MOVSX BX, AL”执行后, BX=_____。
 - (8) 将 EBX 内无符号整数乘以 8, 可以使用一条移位指令“_____ EBX, _____”, 其中后一个操作数是一个立即数。

(9) 在 MASM 源程序中, 要使用 Pentium Pro 支持的新增指令, 需要书写处理器选择伪指令_____。如果该程序将运行于虚拟 DOS 平台, 则应书写在存储模型伪指令之_____ (选择: 前或后)。

(10) 使用 DD 伪指令定义的一个变量 VAR, 其类型名是_____, TYPE VAR 等于_____。

6.4 简述 Intel 80x86 系列处理器在指令集方面的发展。

6.5 什么是平展存储模型、段式存储模型和实地址存储模型?

6.6 什么是实地址方式、保护方式和虚拟 8086 方式? 它们分别使用什么存储模型?

6.7 说明下列指令如何计算存储器操作数的单元地址, 其中 VAR 是一个变量名:

(1) `add [ebx+8*ecx],al`

(2) `mov var[ecx+ebx],cx`

(3) `sub eax,dword ptr var`

(4) `mov ecx,[ebx]`

6.8 思考 LEA 指令的功能, 然后用一条 LEA 指令实现如下运算操作:

$EAX \leftarrow EBX + ESI \times 2 + 1234H$

能够保证该运算正确的条件是什么?

6.9 阅读如下程序片段, 为每条语句加上注释, 并说明该程序片段的功能。其中 ARRAY 是一个双字类型的数组。

```
xor eax,eax
mov ebx,offset array
mov ecx,3
mov eax,[ebx+4*ecx]
mov ecx,5
add eax,[ebx+4*ecx]
mov ecx,7
add eax,[ebx+4*ecx]
```

6.10 完成下列要求的程序片段:

(1) 选择一条指令完成将 EBX 的内容减 1。

(2) 将 EAX、EBX、ECX 内容相加, 并将和存入 EDX 寄存器。

(3) 请用两种方法都只用一条指令实现 EAX 乘以 16。

6.11 有两个 64 位无符号整数存放在变量 buffer1 和 buffer2 中, 定义数据、编写代码完成 $EDX.EAX \leftarrow \text{buffer1} - \text{buffer2}$ 功能。

6.12 请用 80386 处理器的指令实现如下 80486 的新增指令, 并写成宏结构形式:

(1) BSWAP

(2) XADD

(3) CMPXCHG

6.13 参考例 5-4 的算法, 使用 32 位寄存器表达数据, 实现以十进制形式显示 32 位二进制数的子程序, 并用于例 6-3 的数据显示中。

6.14 由于采用 16 位指令, 第 4 章习题 4.29 和习题 4.30 只实现 N 小于 65535 的素数判断和个数计算。现使用 32 位指令实现更大 N 值的素数判断和个数计算。

第 7 章 Windows 编程

所谓 Windows 编程是指直接利用 Windows 系统函数编写应用程序，即直接调用应用程序接口 API（Application Program Interface）函数进行编程。本章介绍 Windows 的基本控制台函数和图形窗口函数，引出 MASM 为支持高级语言所提供的特性，详解使用汇编语言的高级语言特性调用 API 函数的方法。

经过多年的发展，Windows 存在 16 位、32 位和 64 位多种版本。本章基于 32 位 Windows，使用 32 位指令，可以认为是第 6 章 32 位指令编程的继续，只是操作系统平台将微软 16 位的 DOS 换成了微软 32 位的 Windows。另外，为了更好地调用 Windows 函数，本章还将介绍 MASM 的高级特性，也是比较复杂和深入的内容。

7.1 操作系统函数调用

高级语言支持许多标准函数，其集成开发环境还提供增强功能。所以，开发基于 Windows 平台的应用程序，尤其是图形窗口程序，程序员自然会选择简单实用、功能强大的多种可视化环境，如 Visual Basic、Visual C++ 等。汇编语言也可以编写 32 位 Windows 应用程序，不过，汇编程序并没有标准函数可以利用，必须通过操作系统提供的功能实现。

Windows 的系统函数（功能）以动态连接库 DLL（Dynamic-Link Library）形式提供，利用其应用程序接口 API 调用动态连接库中的函数。API 是一些类型、常量和函数的集合，提供编程中使用库函数的途径。Windows 的 API 也被称为软件开发包 SDK（Software Development Kit）。16 位 Windows 的 API 被称为 Win16；32 位 Windows 的 API 被称为 Win32，兼容 Win16。

7.1.1 动态链接库

为了避免重复编写代码，程序员常把需要重复使用的子程序（或称过程、函数、模块、代码）放到一个或多个库文件（文件扩展名是 .LIB）中。在需要使用这些子程序时，只要把这些库文件和目标文件相连即可。连接程序会自动从这些库文件中抽取需要的子程序插入到最终的可执行代码中，这个过程称为静态链接。应用程序运行时不再需要这些库文件，如前面利用库管理软件生成的子程序库文件，以及 C 语言中的运行库。这种方法的主要缺点是同一个子程序可能被许多应用程序所包含，浪费磁盘空间。

DOS 操作系统是一个单任务操作系统，主存中只有一个程序运行，采用静态链接时主存浪费不太突出。但在多任务操作系统 Windows 中，同一个子程序可能被多个程序或同一个程序多次使用，如果每次调用都占用主存空间，显然浪费就相对严重。为此，提出了动态链接库（文件扩展名是 .DLL）的解决方法。

动态链接库也是保存需要重复使用的代码的文件。但只有运行程序使用它们的时候，Windows 才会将其加载到主存，同时有多个程序使用或者同一个程序多次使用时，主存也只有一份副本。不过，因为应用程序并不包含动态链接库中的代码，所以运行时系统中必须包含该动态链接库，而且该动态链接库文件必须在当前目录或可以搜索到的目录中；否则，程

序将提示没有找到动态链接库文件而无法运行。如果是程序员自己开发的动态链接库，应用程序安装时必须将该动态链接库文件复制到用户机器中。

动态链接库是 Windows 操作系统的基础，Windows 所有的 API 函数都包含在 DLL 文件中。其中有 3 个最重要的系统动态链接库文件，大多数常用函数都存于其中：

- ⊙ KERNEL32.DLL——系统服务函数，主要处理内存管理和进程调度。
- ⊙ USER32.DLL——用户接口函数，主要控制用户界面。
- ⊙ GDI32.DLL——图形设备函数，主要负责图形方面的操作。

如果系统函数不在这 3 个主要库文件中，可以参考微软文档资料，文档将会说明函数在哪个库文件中。早期的 API 文档可以参看 Microsoft Win32 Programmer's Reference (Win32 程序员手册)。最常用的电子文档是一个帮助文件：WIN32.HLP。现在 Windows API 不再以印刷形式出现，只有通过 CD-ROM 光盘或互联网获得电子文档，如利用微软 Windows 程序开发的资料库 MSDN (Microsoft Developer Network, 网址为 <http://msdn.microsoft.com>)。

当需要使用某个 API 函数时，就可以从上述有关资料查找。如果查到它在某个动态链接库中，那么一方面要对这些函数进行过程声明，另一方面需要链接同名的导入库文件（运行时不需要），否则在编译时就会出现 API 函数未定义的错误。

一个动态链接库 DLL 文件，对应一个导入库 (Import Library, 文件扩展名是 .LIB) 文件，如上述 3 个系统动态链接库文件的导入库文件依次是 KERNEL32.LIB、USER32.LIB、GDI32.LIB。之所以还需要导入库文件，是因为动态链接库中的 API 代码本身并不包含在 Windows 可执行文件中，而是当要使用时才被加载。为了让应用程序在运行时能找到这些函数，就必须事先把有关的重定位信息嵌入到应用程序的可执行文件中。这些信息存在于对应的导入库文件中，由链接程序把相关信息从导入库文件中找出并插入到可执行文件中。当应用程序被加载时 Windows 会检查这些信息，这些信息包括动态链接库的名字和其中被调用的函数名。若检查到这样的信息，Windows 就会加载相应的动态链接库。

简单地说，使用 Windows 系统函数，首先需要明确其所在的库文件。在程序开发时需要利用其导入库文件，在运行时需要载入其动态链接库文件。导入库文件存在于程序设计语言的开发环境中，动态链接库存在于操作系统中。但是，单独的 MASM 汇编程序开发环境并没有导入库文件，所以需要借用其他开发环境（如 Visual C++ 集成开发环境）的导入库文件。配套本书的开发环境将几个基本的导入库文件保存于 BIN32 子目录中。

7.1.2 MASM 的过程声明和调用

Windows 的应用程序接口 API 采用 C、C++ 语言语法定义，不便于汇编语言调用。但微软汇编程序从 MASM 6.0 版本开始引入了高级语言具有的程序设计特性（详见 7.3.3 节），为了配合调用高级语言函数，引入了过程声明 PROTO 和过程调用 INVOKE 伪指令。

1. 过程声明伪指令 PROTO

PROTO 用于事先声明过程的结构，包括外部的操作系统 API 函数、高级语言的函数。它的格式如下：

过程名 proto[调用距离][语言类型][[参数]: 类型]...

其中，过程名可以用 PROC 定义的过程名，也可以是 API 函数名或高级语言的函数名。调用距离是指 NEAR 或 FAR 类型，通常默认由存储模型确定。

语言类型有 STDCALL（对应 Windows 系统 API 的调用规范）、C（对应 C 语言使用的调用规范）等。如果该过程使用的语言类型与存储模型 MODEL 伪指令定义的相同，则可以省略，否则必须说明（调用规范的详解见 7.2 节）。

PROTO 语句最后是该过程带有的参数以及类型，冒号前的参数名可以省略，但冒号和类型不能省略。类型可以使用任何 MASM 有效的类型：对于变量可以是 DWORD、WORD、BYTE 等；对于地址可以用 DWORD（32 位地址）或 WORD（16 位地址）说明，也可以是 PTR，说明为指针（为了简化，本书转换过程中一律使用 DWORD）；对于参数个数、类型不定，需要使用 VARARG 进行说明（Variable Argument，表示可变参数）。

2. 过程调用伪指令 INVOKE

处理器使用 CALL 指令实现子程序调用。但由于涉及堆栈传递参数的传递规范，直接使用 CALL 调用高级语言函数比较烦琐。另外，经过 PROTO 过程声明的过程或函数，汇编系统会对其进行类型检测，也需要配合使用过程调用伪指令 INVOKE 来实现调用，它的格式如下：

```
invoke 过程名[参数,...]
```

过程调用伪指令自动创建调用过程所需要的代码序列，调用前将参数压入堆栈，调用后平衡堆栈。其中“参数”表示通过堆栈传递给过程的实参数，可以是各种常量组成的数值表达式、通用寄存器、寄存器对（格式是 reg::reg）、标号或变量地址等。

对于地址参数常会使用“ADDR”操作符，后跟标号或者变量名字，表示它们的地址。ADDR 操作符类似 OFFSET 操作符，但 ADDR 只用在 INVOKE 语句中，常用于获取局部变量的地址；而 OFFSET 只能获取全局变量的偏移地址。MASM 中在数据段定义的变量都是全局变量。局部变量使用 LOCAL 伪指令定义，占用堆栈区域（详见 7.2 节），需要使用 ADDR，而不能使用 OFFSET 来获取地址。

使用 INVOKE 调用之前，需要用 PROTO 先进行声明，或者先用扩展的 PROC 进行过程定义（详见 7.3 节）。

7.1.3 程序退出函数

程序执行结束需要退出，Windows 使用 ExitProcess 函数实现（对应 DOS 的 4CH 号功能调用），该函数存在于 32 位核心动态链接库（KERNEL32.DLL）中。它是一个标准的 Windows API，结束一个进程及其所有线程，即程序退出。在 Win32 程序员参考手册中，它的定义如下：

```
VOID ExitProcess(  
    UINT uExitCode    // exit code for all threads  
);
```

其中参数 uExitCode 表示该进程的退出代码，其作用与 DOS 操作系统的退出功能调用（如 4CH 号）的入口参数相同。例如，常用 0 表示程序正常执行结束。

退出代码的类型 UINT 表示无符号整型，而 int 型在 32 位系统中是 32 位整数，所以 UINT 表示 32 位无符号整数类型。在文档中，API 函数的声明采用 C/C++ 语法，所有函数的参数类型都是基于标准 C 语言的数据类型或者 Windows 的预定义类型。我们需要正确地区别这些类型，才能转换成汇编语言的数据类型。例如，类型 UNIT 对应汇编语言的双字类型 DWORD。

注意，汇编语言通常不区分大小写字母，所以经常使用汇编语言可能形成一律使用小写或大写，或者按照一定规律大小写混用（如保留字用大写，其他标识符用小写）的习惯。但是，高级语言 C 和 C++ 对大小写敏感，大写字母与小写字母须严格区分，所以在汇编语言中使用高级语言的函数、类型或者变量等标识符，都要按照高级语言的语法正确书写大小写，不能像汇编语言那样随意。

这样，ExitProcess 函数在汇编语言中，需要进行如下声明：

```
ExitProcess proto ,:dword
```

应用程序中使用该功能，这个应用程序就会立即退出，返回 Windows。汇编语言的调用方法如下：

```
invoke ExitProcess,0
```

其中，返回代码是 0，表示没有错误。返回代码也可以是其他数值。

利用 MASM 的 PROTO 和 INVOKE 语句，不仅可以在调用函数时对函数声明的原型进行类型检测，以便发现是否有参数不匹配的情况；而且汇编语言中调用 Windows 的 API 函数就像 C/C++ 等高级语言一样。

还可以利用 MASM 的宏汇编能力，将函数调用定义成宏。例如：

```
exit      macro dwexitcode
            invoke ExitProcess,dwexitcode
        endm
```

利用这个宏实现程序退出，宏指令是：

```
exit 0
```

这样使用，更加简单方便。

7.1.4 Windows 程序格式

类似 DOS 环境的汇编语言程序，Windows 应用程序的格式如下：

```
;exampled.asm in Windows
.686
.model flat,stdcall
option casemap:none
includelib bin32\kernel32.lib ;包含基本 API 函数的导入库文件
ExitProcess proto,:dword      ;Windows 函数声明
.data                          ;定义数据段
.....                        ;数据定义(数据待填)
.code                          ;定义代码段
start:                          ;程序执行起始位置
.....                          ;主程序(指令待填)
invoke ExitProcess,0           ;程序正常执行终止
.....                          ;子程序(指令待填)
end start                      ;汇编结束
```

编写 32 位 Windows 应用程序，肯定要使用 32 位指令，所以必须有处理器选择伪指令，这里使用“.686”声明采用 Pentium Pro（原被称为 80686 处理器）支持的指令系统。还应注意，该伪指令必须书写在存储模型语句之前，默认采用 32 位地址和操作数长度。

32 位 Windows 使用线性地址空间，存储模型语句“.MODEL”只能选择 FLAT 平展模型。程序需要使用 Windows 提供的系统函数，它的应用程序接口 API 采用标准调用语言类型

“STDCALL”。

汇编语言默认不区分大小写，但选项伪指令“OPTION CASEMAP:NONE”告知 MASM 要区分标识符的大小写，因为 Windows 的 API 函数区别大小写。汇编程序 ML.EXE 的参数“/Cp”具有同样的效果，也是告知 MASM 不要更改程序员自己定义的标识符的大小写。这种情况下，虽然 MASM 保留字使用大小写均可，但用户自定义的符号不能随意使用大小写。

本书配套软件包将基本的 Windows 导入库等文件保存在 ML615 主目录的 BIN32 子目录下。如果需要使用其他导入库文件，也需要相应地包含它们。程序中使用的 API 函数也需要使用 PROTO 语句进行声明。

堆栈段通常由 Windows 操作系统维护，用户可以不设置；如果程序使用的堆栈空间较大，也可以设置。同样，Windows 不需要用户设置代码段 CS、数据段 DS 内容，因为它们共用一个平展（不分段）的线性地址空间。

程序格式中定义了一个标号 START（也可以使用其他标识符），汇编结束 END 语句作为参数，用于指明程序开始执行的位置。

可以对比第 1 章引出的 DOS 应用程序格式，从实际编程角度看，同样也是在数据段定义变量、常量等，在代码段书写程序代码。

7.2 控制台应用程序

当一个 Windows 应用程序开始运行时，它可以创建一个控制台（Console）窗口，也可以创建一个图形界面窗口。32 位 Windows 控制台程序看起来像一个增强版的 MS-DOS 程序，如它们都使用标准的输入设备（键盘）和输出设备（显示器）。但实质上，32 位控制台程序完全不同于 MS-DOS 程序，因为它运行在保护方式下，通过 API 使用 Windows 的动态链接库函数。

7.2.1 控制台输出

编写控制台程序需要调用控制台函数，实现基本的控制台输出（显示器）、控制台输入（键盘）。几乎所有的控制台函数都要求将控制台句柄作为第一个参数传递给它们。

本节介绍的控制台函数都存放在 KERNEL32.DLL 动态库中，程序开发过程中需要使用 KERNEL32.LIB 导入库文件。

1. 控制台句柄

句柄是一个 32 位无符号整数，用来确定一个唯一对象，例如某个输入设备、输出设备或者一个文件、图形等。

Windows 为每种标准设备定义了一个常量句柄，它们是：标准输入句柄（常量符号：STD_INPUT_HANDLE，数值：-10）、标准输出句柄（常量符号：STD_OUTPUT_HANDLE，数值：-11）和标准错误句柄（常量符号：STD_ERROR_HANDLE，数值：-12）。标准错误设备的默认输出位置也是显示器。在汇编语言中可以如下定义：

```
STD_INPUT_HANDLE  = -10
STD_OUTPUT_HANDLE = -11
STD_ERROR_HANDLE  = -12
```

在控制台程序中进行任何的输入输出操作都需要首先使用获取句柄函数 `GetStdHandle` 来获得一个句柄实例。该函数原型定义如下：

```
HANDLE GetStdHandle(
    DWORD nStdHandle    // input, output, or error device
);
```

所以，`GetStdHandle` 函数在汇编语言中可以声明如下：

```
GetStdHandle    proto,nStdHandle:dword
```

其中，`nStdHandle` 参数（在声明中可以省略这个参数名，也可以是其他名，但后面的类型不能省略）可以是标准输入、标准输出或标准错误。例如，获得标准输出的句柄实例为：

```
invoke GetStdHandle,STD_OUTPUT_HANDLE
```

API 函数的返回值保存于 `EAX` 中。所以，`GetStdHandle` 函数执行结束时，`EAX` 寄存器返回一个句柄实例（即 `HANDLE` 定义的双字类型）。为了以后使用，应该把它保存起来。

对标准输入和标准输出设备，一个程序中获得一个句柄就可以了。

2. 控制台输出函数

在控制台环境常用 API 函数 `WriteConsole` 来实现显示器输出，使用控制台输出句柄实例将一个字符串输出到屏幕上，并支持标准的 ASCII 控制字符，如回车、换行等。

Win32 API 中可以使用两种字符集：美国国家标准学会 ANSI 定义的 8 位的 ASCII 字符集和 16 位的 Unicode 字符集。用于文本操作的 Win32 API 函数往往有两个不同版本。8 位 ANSI 字符集的版本中，函数名以字母 A 结尾（如 `WriteConsoleA`）；16 位宽字符集（包括 Unicode 字符集）的版本中，函数名以字母 W 结尾（如 `WriteConsoleW`）。

Windows 95/98 操作系统不支持以 W 结尾的函数。Windows NT/2000/XP 操作系统的内置字符集是 Unicode，在这些操作系统中调用以 A 结尾的函数时，操作系统首先会将 ANSI 字符转换成 Unicode 字符，再调用对应以 W 结尾的函数。

在 MSDN 文档中，函数名尾部的字母 A 或 W 被省略（如 `WriteConsole`）。汇编语言可以利用等价伪指令重新定义函数名：

```
WriteConsole    equ <WriteConsoleA>
```

这样就可以通过正常的函数名来调用 `WriteConsole` 函数。

`WriteConsole` 函数原型定义如下：

```
BOOL WriteConsole(
    HANDLE hConsoleOutput,    // handle to a console screen buffer
    CONST VOID *lpBuffer,     // pointer to buffer to write from
    DWORD nNumberOfCharsToWrite, // number of characters to write
    LPDWORD lpNumberOfCharsWritten, // pointer to number of characters written
    LPVOID lpReserved         // reserved
);
```

原型中的大写字符串表示参数类型，有逻辑型（`BOOL`）、句柄（`HANDLE`）、指针、整型等，根据含义，大多数都对应汇编语言的 `DWORD` 类型。

`WriteConsole` 函数在汇编语言中可以如下声明：

```
WriteConsoleA proto,
    handle:dword,    ;输出句柄
    pBuffer:dword,   ;输出缓冲区指针
    bufsize:dword,   ;输出缓冲区大小
```

```
pCount:dword,          ;实际输出字符数量的指针
lpReserved:dword      ;保留（必须为 0）
```

第 1 个参数是控制台输出句柄实例；第 2 个参数是指向字符串的指针，即缓冲区地址；第 3 个参数指明字符串长度，是一个 32 位整数；第 4 个参数指向一个整数变量，函数运行结束将在这里返回实际输出的字符数量；最后一个参数保留，使用时必须设置为 0。注意，控制台输出函数 WriteConsole 要求知道显示字符串的字符个数，不使用结尾字符。

WriteConsole 函数执行成功，返回一个非 0 值；如果没有正确执行，则返回值为 0。

【例 7-1】 控制台输出程序。

本例程序直接调用 Windows 控制台 API 函数显示欢迎信息，请对比例 1-1 程序。

```
.686
.model flat,stdcall
option casemap:none
includelib bin32\kernel32.lib    ;包含 API 函数的导入库文件
ExitProcess proto,:dword         ;Windows 函数声明
GetStdHandle proto,:dword
WriteConsoleA proto,:dword,:dword,:dword,:dword,:dword
WriteConsole equ <WriteConsoleA>
STD_OUTPUT_HANDLE = -11         ;Windows 常量定义

.data
msg db 'Hello, Assembly!',13,10 ;字符串
outsize dd ?                    ;保存实际输出的字符数量

.code

start:
    ;获得输出句柄
    invoke GetStdHandle,STD_OUTPUT_HANDLE
    ;显示信息
    invoke WriteConsole,eax,addr msg,sizeof msg,addr outsize,0
    ;退出
    invoke ExitProcess,0
end start
```

汇编语言使用高级语言函数的一般步骤如下：

- （1）使用 INCLUDELIB 伪指令包含对应函数的导入库文件。
- （2）使用 PROTO 伪指令进行原型声明，包括形参类型。
- （3）使用 INVOKE 伪指令调用函数，代入实参。

另外，在汇编语言中使用 API 函数时，应特别注意一个问题。通常 API 函数使用 EAX 返回参数，但并不保护 EBX、ECX 和 EDX。所以，如果 EAX、EBX、ECX 或 EDX 需要在 API 函数调用后保持不变，应该在调用前进行保护。也可以简单地使用 PUSHAD 保护所有通用寄存器，用 POPAD 恢复所有通用寄存器。

3. 开发过程

开发 Windows 应用程序需要进入 Windows 控制台，可以类似 DOS.BAT 编辑一个批处理文件 WIN.BAT，内容如下：

```
@echo off
@set PATH=ML615;ML615\BIN32;%PATH%
```

```
%SystemRoot%\system32\cmd.exe
@echo on
```

在 Windows 资源管理器中双击该批处理文件,可执行 CMD.EXE 文件快速进入开发环境。在命令行环境,仍然利用 ML.EXE 进行汇编,命令如下:

```
ml /c /coff eg701.asm
```

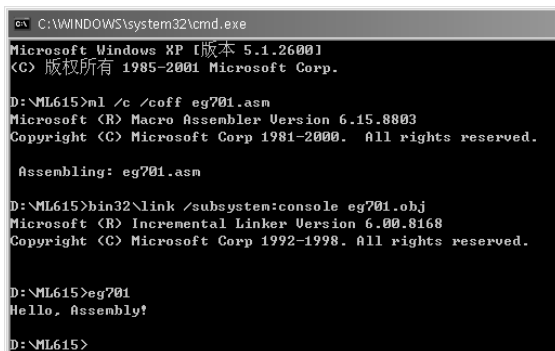
参数 “/coff” (小写字母) 表示生成 COFF (Common Object File Format) 格式的目标模块文件,COFF 是 32 位 Windows 和 UNIX 操作系统使用的目标文件格式。同时,仅进行汇编的参数 “/c” 不能缺少,参数之间一定要用空格分隔。

针对 COFF 格式的目标文件,需要采用增量连接器 (Incremental Linker) 进行连接,而 DOS 使用的连接程序是所谓段式可执行文件连接器 (Segmented Executable Linker)。本书配套软件包将 32 位 Windows 使用的连接程序 (LINK.EXE) 与其他开发 32 位 Windows 程序的导入库等文件一起存放在 BIN32 子目录中,需要使用如下命令:

```
bin32\link /subsystem:console eg701.obj
```

其中,参数 “/subsystem:console” 必须有,表示生成 Windows 控制台 (Console) 环境的可执行文件。如果生成图形窗口的可执行文件,则应使用参数 “/subsystem:windows”。注意,LINK 之前要添加目录 BIN32,以区别同名的 16 位连接程序 (LINK.EXE)。

如果汇编、连接过程没有错误,生成可执行文件,可输入文件名执行程序,如图 7-1 所示。



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

D:\ML615>ml /c /coff eg701.asm
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.

Assembling: eg701.asm

D:\ML615>bin32\link /subsystem:console eg701.obj
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

D:\ML615>eg701
Hello, Assembly!

D:\ML615>
```

图 7-1 Windows 程序的开发过程

7.2.2 控制台输入

ReadConsole 函数是控制台环境常用的键盘输入 API 函数,它将键盘输入的文本保存到一个缓冲区。它支持行内编辑,如使用退格键删除刚输入的字符、使用 ESC 键取消刚输入的所有字符等,最后用 Enter 键确认。

ReadConsole 函数原型定义如下:

```
BOOL ReadConsole(
    HANDLE hConsoleInput,           // handle of a console input buffer
    LPVOID lpBuffer,                // address of buffer to receive data
    DWORD nNumberOfCharsToRead,     // number of characters to read
    LPDWORD lpNumberOfCharsRead,    // address of number of characters read
    LPVOID lpReserved               // reserved
);
```


汇编语言中的声明如下：

```
ReadConsoleA proto,  
                handle:dword,      ;控制台输入句柄  
                pBuffer:dword,     ;输入缓冲区指针  
                maxsize:dword,     ;要读取字符的最大数量  
                pBytesRead:dword,  ;实际输入字符数量的指针  
                notUsed:dword      ;未使用（保留，必须是 0）  
  
ReadConsole equ <ReadConsoleA>
```

当调用这个函数时，系统等待用户输入（如用户输入了 3 个字符，依次是 1、2、3）、输入结束后回车确认。因为回车键代表回车字符 0DH 和换行字符 0AH，所以 pBytesRead 变量保存用户输入的字符个数加 2 的结果（本例的结果为 5，用十六进制数表达依次是 31、32、33、0D、0A）。因此，读者不要忘记在定义输入缓冲区时留出额外的 2 字节。

【例 7-2】 信息输入输出程序。

本例利用控制台输入 ReadConsole 函数和控制台输出 WriteConsole 函数分别编写键盘输入 READMSG 和显示器输出 DISPMSG 子程序，便于调用并实现信息输入和输出的交互。

```
.686  
.model flat,stdcall  
option casemap:none  
includelib bin32\kernel32.lib  
  
ExitProcess proto,:dword  
exit macro dwexitcode  
        invoke ExitProcess,dwexitcode  
        endm  
  
GetStdHandle proto,:dword  
WriteConsoleA proto,:dword,:dword,:dword,:dword,:dword  
WriteConsole equ <WriteConsoleA>  
ReadConsoleA proto,:dword,:dword,:dword,:dword,:dword  
ReadConsole equ <ReadConsoleA>  
STD_INPUT_HANDLE = -10  
STD_OUTPUT_HANDLE = -11  
  
.data  
msg1 db 'Please enter your name: ',0  
msg2 db 'Welcome ',0  
nbuf db 80 dup(0)  
msg3 db ' to Win32 Console!',0  
  
.code  
start: mov eax,offset msg1      ;提示输入  
        call dispmsg  
        mov eax,offset nbuf     ;输入信息  
        call readmsg  
        mov eax,offset msg2  
        call dispmsg  
        mov eax,offset nbuf     ;显示输入信息  
        call dispmsg  
        mov eax,offset msg3
```

```

call dispmsg
exit 0

.data
;子程序 DISPMSG 使用的变量
_outsize dd ?
_outhandle dd ?
.code
dispmsg proc ;字符串显示子程序，入口参数：EAX=字符串地址
push eax
push ebx
push ecx
push edx
push eax ;保存入口参数，即字符串地址
invoke GetStdHandle,STD_OUTPUT_HANDLE
mov _outhandle,ebx ;句柄实例保存，以便后面使用
pop ebx ;从堆栈弹出字符串地址送 EBX
xor ecx,ecx ;计算字符串长度
dispm1: mov al,[ebx+ecx]
test al,al
jz dispm2
inc ecx
jmp dispm1
dispm2: invoke WriteConsole,_outhandle,ebx,ecx,addr _outsize,0
pop edx
pop ecx
pop ebx
pop eax
ret
dispmsg endp

.data
;子程序 READMSG 使用的变量
_insize dd ?
_inbuffer db 255 dup(0) ;设置输入缓冲区最大 255 个字符
.code
readmsg proc ;字符串输入子程序，入口参数：EAX=缓冲区地址
push ebx
push ecx
push edx
push eax ;保护输入的缓冲区地址参数
invoke GetStdHandle,STD_INPUT_HANDLE
invoke ReadConsole,eax,addr _inbuffer,255,addr _insize, 0
sub _insize,2 ;实际输入的字符不包括回车和换行字符
xor ecx,ecx
pop ebx ;获得缓冲区地址
readm1: mov al,_inbuffer[ecx]
mov [ebx+ecx],al ;将输入的字符串复制到用户缓冲区
inc ecx

```

```

        cmp ecx,_inssize
        jnb readm1
        mov byte ptr [ebx+ecx],0 ;最后填入结尾字符 0
        mov eax,ecx             ;返回实际的字符个数，不含结尾标志 0
        pop edx
        pop ecx
        pop ebx
        ret
readmsg   endp
          end start

```

字符串输入子程序 READMSG 涉及两个缓冲区，一个是 ReadConsole 使用的内部缓冲区（_inbuffer），另一个是调用程序设置的用户缓冲区（本例为 nbuf）。该子程序在输入的字符串末尾处添加结尾标志 0，返回实际的字符个数（不包括回车、换行和结尾标志），以方便调用。源程序使用多个数据段定义语句，以便比较清楚地说明定义的变量在主程序或子程序如何使用，但实际上 MASM 汇编程序是将它们组合在一起的，属于同一个数据段。

Windows 控制台还有很多 API 函数，有兴趣的读者可以参考 MSDN 文档，本书不再深入。

7.3 图形窗口应用程序

Windows 图形界面通过窗口、对话框、菜单、按钮等来实现用户交互。使用汇编语言编写图形窗口应用程序就是调用这些 API 函数。

7.3.1 消息窗口

消息窗口是常见的图形显示形式。创建 Windows 的消息窗口非常简单，调用 MessageBox 函数即可，其代码存放在 USER32.DLL 动态链接库中。

MessageBox 是一个标准的 API 函数，其功能是在屏幕上显示一个消息窗口。在 Win32 程序员参考手册中，它的定义如下：

```

int MessageBox(
    HWND hWnd,           // handle of owner window
    LPCTSTR lpText,       // address of text in message box
    LPCTSTR lpCaption,    // address of title of message box
    UINT uType            // style of message box
);

```

其中，hWnd 是父窗口的句柄。如果该值为 NULL（即 0），则说明该消息窗没有父窗口。这里的句柄是窗口的一个地址指针，代表一个窗口。对该窗口进行任何操作，都必须引用该窗口的句柄。

lpText 是要显示字符串的地址指针，即字符串的首地址。lpCaption 为消息窗标题的地址指针。这些字符串都需要以 NULL 结尾，与 C 和 C++ 语言类似。

uType 是一组位标志，指明该消息窗的类型。若该值为 MB_OK（即 0），则该消息窗只具有一个按钮 OK，也是默认值。若该值为 MB_OKCANCEL（即 1），则该对话框有两个按钮 OK 和 Cancel。在中文 Windows 环境下，分别对应中文按钮“确定”和“取消”。

如果未能创建消息窗口，MessageBox 函数返回整数 0。若函数调用成功，则返回用户操

作的菜单项数值，如返回 IDOK 表示用户单击了“确认”按钮。

【例 7-3】 消息窗口程序。

```
.686
.model flat,stdcall
option casemap:none
includelib bin32\kernel32.lib
includelib bin32\user32.lib

ExitProcess    proto,:dword
MessageBoxA    proto :dword,:dword,:dword,:dword
MessageBox     equ <MessageBoxA>
NULL           equ 0
MB_OK          equ 0

.data
szCaption      db '欢迎',0
outbuffer       db '你好，汇编语言!',0

.code

start:

    invoke MessageBox,NULL,addr outbuffer,addr szCaption,MB_OK
    invoke ExitProcess,NULL
end start
```

本例程序在进行连接时需要使用参数“/subsystem:windows”，替代创建控制台程序时使用的参数“/subsystem:console”。这样，汇编连接后将生成一个消息窗口程序。双击之，就运行该程序，弹出消息窗口，标题为“欢迎”、窗口信息为“你好，汇编语言！”，如图 7-2 所示。

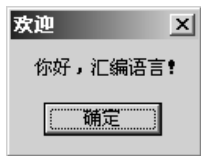


图 7-2 例 7-3 消息窗口程序的运行结果

当然，这只是一个最简单的图形界面程序，还不能说是一个标准的图形窗口程序。编写一个标准 32 位 Windows 图形窗口程序不仅需要熟悉更多的 API 函数，还需要补充一些 MASM 高级特性。所以，接下来的两节补充介绍 MASM 高级特性，并借助一个较完整的免费软件开发包 MASM32 来了解开发图形窗口应用程序的方法。

7.3.2 结构变量

类似于高级语言中的用户自定义复合类型数据，MASM 中也允许将若干个相关的单个变量作为一个组来进行整体数据定义，并通过相应的结构预置语句为变量分配空间。例如，结构（Structure）用于把各种不同类型的数据组织到一个数据结构中，便于处理某些变量。

1. 结构类型的说明

结构类型的说明使用一对伪指令 STRUCT（MASM 5.x 为 STRUC，功能相同）和 ENDS，其格式为：

```
结构名 struct
.....           ;数据定义语句
结构名 ends
```

例如，下述语句说明了学生成绩结构：

student	struct
sid	dd ?
sname	db 'unknown '
Math	db 0
English	db 0
student	ends

结构说明中的数据定义语句给定了结构类型中所含的变量，称为结构字段；相应的变量名称为字段名。一个结构中，可以有任意数目的字段，各字段长度可以不同，可以独立存取，可以有名或无名，可以有初值或无初值。

2. 结构变量的定义

结构说明只是定义了一个框架，并未分配主存空间，必须通过结构预置语句分配主存并初始化。结构预置语句的格式为：

变量名 结构名 <字段初值表>

其中，初值表要用尖括号括起来的用逗号分隔的与各字段类型相同的一组数值（或空）。汇编程序将以初值表中的数值顺序初始化对应的各字段，初值表中为空的字段将保持结构说明中指定的初值。另外，结构说明中使用 DUP 操作符说明的字段不能在结构预置语句中初始化。例如，对应上述结构说明，可以定义如下结构变量：

```

stu1      student  <1,'zhang', 85, 90>
stu2      student  <2,'wang',,,>
          student  100 dup( <> )      ;预留 100 个结构变量空间

```

3. 结构变量及其字段的引用

引用结构变量时，直接书写结构变量名即可；引用其中的某个字段时，则需要采用圆点“.”操作符，其格式为：结构变量名.结构字段名。例如：

```
mov stu1.math,95      ;执行指令后，将 math 域的值更新为 95
```

在没有变量名或无法使用变量名时需要采用结构名引用其字段。例如，通过 EBX 指向上述 STUDENT 某个的结构变量，则访问其 SID 字段可使用如下指令：

```
mov [ebx].student.sid,eax ;在没有变量名时要采用结构名引用其字段
```

【例 7-4】系统时钟显示程序。

Windows 具有获取系统时间的 API 函数 GetLocalTime，代码存放于 KERNEL32.DLL 动态链接库中，其 C 语言原型是：

```

VOID GetLocalTime(
    LPSYSTEMTIME lpSystemTime    // address of system time structure
);

```

其中，lpSystemTime 是指向系统时间结构变量 SYSTEMTIME 的指针，SYSTEMTIME 各个字段表明当前年、月、星期、日以及时、分、秒和毫秒。

利用这个函数，使用消息窗口函数显示当前时间（为简化问题，仅显示时、分、秒）。

```

.686
.model flat,stdcall
option casemap:none
includelib bin32\kernel32.lib
includelib bin32\user32.lib

```

```

ExitProcess      proto, :dword
MessageBoxA      proto :dword, :dword, :dword, :dword
MessageBox       equ <MessageBoxA>
                ;系统时间的结构类型说明
SYSTEMTIME      struct
    wYear        dw ?          ;年 (4 位数)
    wMonth       dw ?          ;月 (1~12)
    wDayOfWeek   dw ?          ;星期 (0~6, 0=星期日, 1=星期一, ……)
    wDay         dw ?          ;日 (1~31)
    wHour        dw ?          ;时 (0~23)
    wMinute      dw ?          ;分 (0~59)
    wSecond      dw ?          ;秒 (0~59)
    wMilliseconds dw ?          ;毫秒 (0~999)
SYSTEMTIME      ends
                ;函数声明, 参数是指向结构变量的指针, 也可以用 PTR SYSTEMTIME
GetLocalTime     proto, :dword
writedec         macro time      ;将二进制数转换为两位十进制数, 再转为 ASCII 码保存
    mov ax, time
    mov cl, 10
    div cl        ;商 AL 是百位, 余数 AH 是个位
    add ax, 3030h ;转换为 ASCII 码
    mov [ebx], ax ;对应显示顺序, 百位先显示, 保存在低地址位置
    endm

                .data
mytime          SYSTEMTIME <>          ;系统时间的结构变量定义
timestring      db '--:--:--', 0
timecaption     db '当前时间', 0

                .code
start:
    invoke GetLocalTime, addr mytime    ;获得当前时间
    mov ebx, offset timestring          ;EBX 指向“时”的保存位置
    writedec mytime.wHour                ;转换为 ASCII 码
    add ebx, 3                           ;EBX 指向“分”的保存位置
    writedec mytime.wMinute              ;
    add ebx, 3                           ;EBX 指向“秒”的保存位置
    writedec mytime.wSecond
    invoke MessageBox, 0, addr timestring, addr timecaption, 1
    invoke ExitProcess, 0
    end start

```

MASM 除具有定义简单数据类型的伪指令（DB、DW、DD 等）外，还有上面介绍的结构，以及联合、记录等复杂数据类型的定义伪指令。

MASM 中的联合（UNION）用于为不同的数据类型赋予相同的存储地址，以达到共享的目的；记录（RECORD）提供直接访问数据中若干位的方法，其基本单位是二进制位。另

外，MASM 还提供了“类型定义 TYPEDEF”伪指令，用于创建一个新数据类型，即为已定义的数据类型取一个同义的类型名。这些都与高级语言的相应数据类型类似。

7.3.3 MASM 的高级语言特性

分支、循环和子程序是基本的程序结构，但是用汇编语言编写时却很烦琐、容易出错。为了克服这些缺点，MASM 6.0 开始引入高级语言具有的程序设计特性，即分支和循环的流程控制伪指令，扩展带参数能力的过程定义、声明和调用伪指令，使得可以像高级语言一样来编写分支、循环和子程序结构，能大大减轻汇编语言编程的工作量。

1. 扩展的过程定义

汇编语言中子程序之间和模块之间，利用堆栈进行参数传递都是一个重要和主要的方式。但是，利用堆栈传递参数，相对来说比较复杂且容易出错。为此，MASM 6.x 参照高级语言的函数形式扩展了 PROC 伪指令的功能，使其具有带参数的能力，极大地方便了过程或函数间参数的传递，也方便与高级语言接口实现混合编程（详见 8.2 节）。

在 MASM 6.x 中，带有参数的过程定义伪指令 PROC 格式如下：

```
过程名    proc    [调用距离] [语言类型] [作用范围] [<起始参数>]
                [USES 寄存器列表] [,参数：类型]...
                local 参数表
                ..... ;汇编语言语句
过程名    endp
```

其中，过程所具有的各选项参数如下：

- ④ 过程名——表示该过程的名称，遵循相应语言类型的标识符。
- ④ 调用距离——可以是 NEAR 或 FAR，表示该过程是近或远调用。简化段定义格式中，默认值由.MODEL 语句选择的存储模型决定。
- ④ 语言类型——确定该过程采用的命名约定和调用约定，可以省略（表示与.MODEL 伪指令指定的相同，否则必须说明）。MASM 支持的语言类型有 STDCALL、C 等（详见 8.2 节）。
- ④ 作用范围——可以是 PUBLIC、PRIVATE、EXPORT，表示该过程是否对其他模块可见。默认是 PUBLIC，表示其他模块可见；PRIVATE 表示对外不可见；EXPORT 隐含有 PUBLIC 和 FAR，表示该过程应该放置在导出表中（Export Entry Table）。
- ④ 起始参数——采用这个格式的 PROC 伪指令，汇编系统将自动创建过程的起始代码（Prologue Code）和收尾代码（Epilogue Code），用于传递堆栈参数以及清除堆栈等。起始参数表示传送给起始代码的参数；它必须使用尖括号“< >”括起来，多个参数用逗号分隔。
- ④ 寄存器列表——指通用寄存器名，用空格分隔多个寄存器。只要利用“USES 寄存器列表”罗列该过程中需要保存与恢复的寄存器，汇编系统就将自动地在起始代码中产生相应的入栈指令，并对应在收尾代码中产生出栈指令。
- ④ 参数：类型——表示该过程使用的形式参数及其类型。参数的作用范围是当前的过程内，同样的名字可以在多个过程中使用，但不能与全局变量名或标号相同。类型可以指定为任何 MASM 有效的类型或 PTR（表示地址指针）、VARARG（长度可变的参数）。PROC 伪指令中要使用参数，必须定义语言类型（否则，默认使用.MODEL 伪

指令指定的语言类型)。参数前的各个选项采用空格分隔，而使用参数必须用逗号与前面选项分隔，多个参数也用逗号分隔。

如果过程使用局部变量，紧接着过程定义伪指令 PROC，可以采用一条或多条 LOCAL 伪指令说明。它的格式如下：

local 变量名[个数][:类型][...]

其中，可选的“[个数]”表示同样类型数据的个数，类似数组元素的个数。在 16 位段中默认的类型是字 WORD，在 32 位段中默认的类型是双字 DWORD。使用 LOCAL 伪指令说明局部变量后，汇编系统将自动利用堆栈存放该变量，与高级语言一样（详见 8.2 节）。

对于具有参数的过程定义伪指令，采用 CALL 指令进行调用就显得比较烦琐。通常需要事先用过程声明伪指令 PROTO 说明，然后再使用过程调用伪指令 INVOKE 调用（参见前节）。

【例 7-5】 使用扩展过程定义编写求有符号数平均值程序。

例 5-7 求 16 位有符号整型数组元素的平均值的程序中，子程序使用了两个参数：数组指针和元素个数；例 5-7 中使用的堆栈传递入口参数，是扩展带参数过程定义使用的方法，也是高级语言函数进行参数传递的方法。现进行改写，以便对比。注意，这是一个 16 位 DOS 环境的应用程序（不是本章的 32 位 Windows 应用程序），仅使用 16 位 8086 指令，需要像前 5 章那样进行汇编和连接。

```
.model small
.stack
mean    proto c,:word,:word      ;过程声明，使用 C 语言类型
.data
array   dw 675, 354, -34, 198, 267, 0, 9, 2371, -67, 4257
.code
.startup
mov ax,lengthof array      ;主程序：调用求平均值子程序
mov bx,offset array
invoke mean,bx,ax
call dispsiw               ;显示
.exit
;子程序：计算 16 位有符号数平均值
;入口参数：D 表示数组地址、NUM 表示元素个数
;出口参数：AX=平均值
mean    proc c uses bx cx dx,d:word,num:word
mov bx,d                   ;BX=数组指针
mov cx,num                 ;CX=元素个数
xor ax,ax                  ;AX 保存和值
mean1:  add ax,[bx]          ;求和
        add bx,type array    ;指向下一个数据
        loop mean1           ;循环
        cwd                  ;将累加和 AX 符号扩展到 DX
        idiv num              ;有符号数除法，AX=平均值（余数在 DX 中）
        ret
mean    endp
dispsiw proc c uses bx cx dx,d:word,num:word
        .....
        ;同例 5-4，省略
dispsiw endp
end
```


对比例 5-7 子程序 MEAN，使用的参数直接书写在 PROC 的过程定义中，子程序直接使用它们的名称，不必考虑对 BP 的操作；使用 UESE 说明子程序使用的寄存器，子程序也不需要书写入栈和出栈指令。主程序中调用子程序，直接代入参数，也不需考虑堆栈平衡问题。这样编程是不是简单多了？

虽然封装了编程细节，但无法改变传递参数的实质，打开列表文件或者在调试程序查看反汇编代码就能体会了。为了在列表文件中看到这些伪指令生成的处理器指令，需要在汇编时带上“/Sa”参数，表示最大化源代码列表。

打开这样生成的列表文件，重点观察过程定义和调用，代码段的列表内容如下（删除了每个语句行前面的地址和机器代码部分）。

主程序的过程调用代码：

```
        mov ax,lengthof array
        mov bx,offset array
        invoke mean,bx,ax
*       push ax
*       push bx
*       call mean
*       add  sp,00004h
```

子程序的过程定义代码：

```
mean    proc c uses bx cx dx,d:word,num:word
*       push bp
*       mov  bp,sp
*       push bx
*       push cx
*       push dx
        mov bx,d      ;BX=数组指针
        mov cx,num    ;CX=元素个数
        .....        ;这部分同源程序，略
        idiv num
        ret
*       pop  dx
*       pop  cx
*       pop  bx
*       pop  bp
*       ret   00000h
mean    endp
```

列表文件中带星号“*”的语句是汇编程序生成的，此时再对比例 5-7 程序，是不是一样呢？

主程序 INVOKE 语句生成的代码，与使用堆栈传递参数进行 CALL 调用一样。子程序使用 USES 生成的保护和恢复寄存器的代码也一样，保护 BP 并通过 BP 指针访问入口参数也一样。在列表文件后面的符号部分有这两个参数的类型和取值说明：

	Name	Type	Value
d	Word		bp + 0004
num	Word		bp + 0006

如果在本例程序的 PROTO 和 PROC 语句中，使用 STDCALL 语言类型替代 C 语言类型，那么同样汇编和生成列表文件后观察可发现主程序调整 SP 指针的指令“ADD SP, 00004H”

不见了，子程序最后的返回指令变成了“RET 00004H”。

80x86 系列处理器中，不论是什么操作系统平台，利用堆栈传递参数都是最常用的方法。为了对比，将上述程序修改在 32 位 Windows 操作系统平台，并将数组元素也定义为 32 位数据，如下所示。

```
.686
.model flat,stdcall
option casemap:none
includelib bin32\kernel32.lib

ExitProcess proto,:dword
mean        proto c,:dword,:dword      ;过程声明，使用 C 语言规范
.data
array       dd 675,354,-34,198,267,0,9,2371,-67,4257
.code
start:      mov eax,lengthof array
            mov ebx,offset array
            invoke mean,ebx,eax
            invoke ExitProcess,0

mean        proc c uses ebx ecx edx,d:dword,num:dword
            mov ebx,d                  ;EBX=数组指针
            mov ecx,num                ;ECX=元素个数
            xor eax,eax                 ;EAX 保存和值
            xor edx,edx                 ;EDX=指向数组元素
mean1:      add eax,[ebx+edx*4]          ;求和
            add edx,1                  ;指向下一个数据
            cmp edx,ecx                 ;比较个数
            jb mean1                   ;循环
            cdq                         ;将累加和 EAX 符号扩展到 EDX
            idiv ecx                    ;有符号数除法，EAX=平均值（余数在 EDX 中）
            ret
mean        endp
end start
```

在 Windows 控制台汇编、连接该源程序（EG705F.ASM），但应注意在生成列表文件时带上“/Sa”参数，以便详细地罗列相关信息。同样，重点观察过程定义和调用，主程序的过程调用代码：

```
mov eax,lengthof array
mov ebx,offset array
invoke mean,ebx,eax

*   push eax
*   push ebx
*   call mean
*   add esp,000000008h
```

子程序的过程定义代码：

```
mean        proc c uses ebx ecx edx,d:dword,num:dword
*   push ebp
*   mov ebp,esp
*   push ebx
```

```

*   push ecx
*   push edx
      mov ebx,d      ;EBX=数组指针
      mov ecx,num    ;ECX=元素个数
      .....        ;这部分同源程序，略
      ret
*   pop edx
*   pop ecx
*   pop ebx
*   pop ebp
*   ret 00000h
mean   endp

```

可以看到，在 32 位 Windows 平台，调用指令 CALL 将压入 4 字节的 32 位偏移地址，而子程序也将使用 32 位寄存器 EBP 替代 16 位平台的 16 位寄存器 BP。这样，参数数组指针 D 和元素个数 NUM 相对 EBP 的位移量分别是 8 和 12，如下文件说明：

	Name	Type	Value
d	DWord	bp + 00000008	
num	DWord	bp + 0000000C	

2. 条件控制

MASM 6.0 引入 .IF、.ELSEIF、.ELSE 和 .ENDIF 伪指令，它们类似于高级语言中的 IF、THEN、ELSE 和 ENDIF。这些伪指令在汇编时自动生成相应的比较和条件转移指令序列，实现程序分支转移。利用条件控制伪指令可以简化分支结构的编程。

条件控制伪指令的格式如下：

```

.if 条件表达式      ;条件为真（值为非 0），执行分支体
    分支体
[.elseif 条件表达式 ;前面 IF [以及前面 ELSEIF] 条件为假（值为 0），
                    ;并且当前 ELSEIF 条件为真，执行分支体
    分支体 ]
[.else              ;前面 IF [以及前面 ELSEIF] 条件为假，
                    ;执行分支体
    分支体 ]
.endif              ;分支结束

```

其中，方括号内的部分可选；条件表达式允许的操作符参见表 7-1。

表 7-1 条件表达式中的操作符

操 作 符	功 能	操 作 符	功 能	操 作 符	功 能
==	等于	&&	逻辑与	CARRY?	CF=1?
!=	不等于		逻辑或	OVERFLOW?	OF=1?
>	大于	!	逻辑非	PARITY?	PF=1?
>=	大于等于	&	位测试	SIGN?	SF=1?
<	小于	()	改变优先级	ZERO?	ZF=1?
<=	小于等于				

汇编程序在翻译相应条件表达式时，将生成一组功能等价的比较、测试和转移指令。操作符的优先关系为逻辑非“!”最高，然后是表 7-1 中左列的比较类操作符，最低的是逻辑与

“&&”和逻辑或“||”，当然也可以加“()”来改变运算的优先顺序，即先括号内、后括号外。位测试操作符的使用格式是“数值表达式 & 位数”。

条件表达式用“非 0”表示成立（真），用 0 表示不成立（假）。

【例 7-6】 使用条件控制的程序。

(1) 例 4-8 求 AX 绝对值的单分支结构，可编写成：

```
mov ax,var
.if ax < 0
    neg ax          ;满足，求补
.endif
mov result,ax
```

(2) 例 4-10 显示数据最高位的双分支结构，可编写成：

```
mov bx,var
shl bx,1           ;BX 最高位移入 CF 标志
.if carry?
    mov dl,'1'      ;如果 CF=1，设置 DL='1'
.else
    mov dl,'0'      ;否则 CF=0，设置 DL='0'
.endif
mov ah,2
int 21h           ;显示
```

(3) 例 4-9 字母判断略复杂，可编写成：

```
mov ah,1
int 21h           ;输入一个字符，从 AL 返回值
.if al >= 'A' && al <= 'Z' ;是大写字母（A 与 Z 之间）
    or al,20h       ;转换为小写
    mov dl,al
    mov ah,2
    int 21h         ;显示小写字母
.endif
```

将上述源程序分别进行编辑，带上“/Sa”参数汇编、生成列表文件，对比如下（带有“*”的语句是由汇编程序产生的）。

(1) 例 4-8 求 AX 绝对值程序：

```
        .if ax < 0
*      cmp  ax, 000h
*      jae  @C0001
        neg ax          ;满足，求补
        .endif
*@C0001:
```

汇编程序自动创建需要的标号（如“@C0001”），并保证程序当中的唯一性。

(2) 例 4-10 显示数据最高位程序：

```
        .if carry?
*      jae  @C0001
        mov dl,'1'      ;如果 CF=1，设置 DL='1'
        .else
*      jmp  @C0003
```

```

*@C0001:
    mov dl,'0'                ;否则 CF=0, 设置 DL='0'
    .endif

```

```

*@C0003:

```

(3) 例 4-9 字母判断程序:

```

    .if al >= 'A' && al <= 'Z' ;是大写字母 (A 与 Z 之间)
*      cmp  al, 'A'
*      jb   @C0001
*      cmp  al, 'Z'
*      ja   @C0001
    or al,20h                ;转换为小写
    mov dl,al
    mov ah,2
    int 21h                  ;显示小写字母
    .endif
*@C0001:

```

通过运行程序、对比相应的原示例程序, 可发现例 4-8 的修改有问题, 应该生成比较有符号数大小的条件转移指令 JGE, 程序才正确。但为什么产生无符号数比较大小的条件转移指令 JAE 呢?

这是因为, 采用寄存器或常数作为条件表达式的数值参加比较时, MASM 汇编程序默认为无符号数。如果作为有符号数, 可以利用 SBYTE PTR、SWORD PTR 或 SDWORD PTR 等操作符进行强制转换。本例中改进为 “.if sword ptr ax < 0” 即可。

同样, 对于条件表达式中的变量, 若是用 DB (BYTE)、DW (WORD)、DD (DWORD) 等变量定义伪指令, 也是作为无符号数对待。若需要进行有符号数的比较, 这些变量在数据定义时须用相应的带符号数据定义语句来定义, 依次为 SBYTE、SWORD、SDWORD 等, 或者进行强制转换。

如果条件表达式中一个数值为有符号数, 则条件表达式强制另一个数据作为有符号数进行比较。所以, 使用条件控制伪指令时, 要注意条件表达式比较的两个数值是作为无符号数还是作为有符号数, 因为它将影响产生的条件转移指令。其实在高级语言中有时也需要区别无符号整数和有符号整数, 例如 C、C++ 语言默认使用有符号整数, 表示无符号整数时需要使用 unsigned 进行说明。

最后对比说明一下, 这里的条件控制伪指令 (前有一个圆点符号) 与前一章最后的条件汇编伪指令虽然从形式上看很相似, 但实质上并不相同。条件汇编伪指令对于分支体的取舍是静态的, 是在程序执行前的汇编阶段完成的, 执行程序中只含有两个分支中的一支, 程序执行时不再需要条件判断; 而条件控制伪指令组成的程序段对两个分支均要进行汇编, 产生相应的指令并被包含在程序中, 程序执行时再进行相应的条件判断, 从而选择其中一支执行; 它对分支体的取舍是动态的。

3. 循环控制

用处理器指令实现的循环控制结构非常灵活, 但可读性不如高级语言, 容易出错, 一不小心就会将循环与分支混淆。利用 MASM 6.x 提供的循环控制伪指令设计循环程序, 可以简化编程使结构清晰。用于循环结构的流程控制伪指令有: .WHILE 和 .ENDW、.REPEAT

和.UNTIL 以及.REPEAT 和.UNTILCXZ, 另外.BREAK 和.CONTINUE 分别表示无条件退出循环和转向循环体开始。利用这些伪指令可以形成两种基本循环结构形式, 分别是先判断循环条件的 WHILE 结构和后判断循环条件的 UNTIL 结构, 参见图 4-8。

【例 7-7】 使用循环控制的程序。

(1) WHILE 结构的循环控制伪指令的格式为:

```
.while 条件表达式      ;条件为真, 执行循环体
.....                ;循环体
.endw                  ;循环体结束
```

格式中条件表达式与条件控制伪指令 .IF 后跟的条件表达式一样, 不再重复 (下同)。

例如, 实现 $1+2+3+\cdots+100$ 累加和, 可以编写为:

```
xor ax,ax              ;被加数 AX 清 0
mov cx,100
.while cx!=0
    add ax,cx          ;从 100, 99, ....., 2, 1 倒序累加
    dec cx
.endw
```

(2) UNTIL 结构的循环控制伪指令的格式为:

```
.repeat                ;重复执行循环体
.....                ;循环体
.until 条件表达式      ;直到条件为真
```

这样, 实现 $1\sim 100$ 求和, 循环体部分也可以编写为:

```
.repeat
    add ax,cx
    dec cx
.until cx==0
```

(3) UNTIL 结构还有一种格式:

```
.repeat                ;重复执行循环体
.....                ;循环体
.untilcxz [条件表达式] ;(E)CX=(E)CX-1, 直到(E)CX=0 或条件为真
```

不带表达式的 .REPEAT / .UNTILCXZ 伪指令将汇编成一条 LOOP 指令, 即重复执行直到 (E)CX=0。带有表达式的 .REPEAT / .UNTILCXZ 伪指令的循环结束条件是 (E)CX 等于 0 或指定的条件为真。 .UNTILCXZ 伪指令的表达式只能是比较寄存器与寄存器、存储单元和常数, 以及存储单元与常数相等 (==) 或不等 (!=)。在 16 位平台使用 CX、32 位平台使用 ECX 做计数器。

这样, 实现 $1\sim 100$ 求和, 循环体部分还可以编写为:

```
.repeat
    add ax,cx
.untilcxz
```

【例 7-8】 斐波那契数程序。

同例 4-17, 改用循环流程控制伪指令, 实现输出斐波那契数。

```
mov ax,1              ;AX=F(1)=1
call dispuiw          ;显示第 1 个数
call dispCrLf         ;回车换行
call dispuid          ;显示第 2 个数
```

```

call dispctrlf      ;回车换行
mov bx,ax           ;BX=F(2)=1
.while -1           ;无条件循环
    add ax,bx       ;AX=F(N)=F(N-2)+F(N-1)
    .break.if carry? ;如果出现进位，则退出循环
    call dispuiw    ;显示一个数
    call dispctrlf  ;回车换行
    xchg ax,bx      ;AX=F(N-2), BX=F(N-1)
.endw

```

带上“/Sa”参数汇编、生成列表文件，观察汇编程序生成的代码，对比不使用流程控制指令的代码，这也是一个学习的过程。虽然汇编程序生成的代码可读性略差，有时生成的代码也不好，但很多时候的编程技术相当老道、值得体味。

需要特别提醒的是，虽然条件控制伪指令和循环控制伪指令看起来简单好用，但如果不能充分理解其实质，往往会隐藏不易察觉的错误。所以，建议初学者一定要查看其列表文件，注意有符号数、无符号数比较等问题，确保生成的条件转移等指令是正确的。不要因为 MASM 提供了简单易用的高级语言特性，而忽略了对使用处理器控制转移类指令实现分支、循环和子程序结构的学习。因为使用处理器指令编写的程序才是真正的汇编语言程序，是我们需要首先掌握的内容。而使用高级语言特性之后的程序已经不再是“纯粹”的汇编语言程序。当然，在必须需要使用汇编语言编写应用程序时完全可以使用这些高级语言特性，以提高编程效率。

7.3.4 简单窗口程序

利用 API 函数，从最基础开始开发一个 32 位 Windows 应用程序确实不太容易。我们不仅需要掌握各种 API 函数的调用方法，还必须将它们转换为汇编语言的形式进行声明，因为 API 函数都是采用 C/C++ 语法进行声明的。手工翻译这些包含声明的头文件既烦琐又容易出错。微软的软件开发包 SDK 中有一个转换工具 H2INC。利用这个工具能够将 C 风格的头文件转换成 MASM 兼容的包含文件（.INC）。这个工具只能转换常量、结构和函数声明，无法转换 C 代码。

1. MASM32 开发包

当前，虽然没有商业的、专门的利用汇编语言开发 32 位 Windows 应用程序的集成软件包，但 Steve Hutchesson 为我们提供了一个免费软件开发包 MASM32。其中包括编辑器、MASM 6.14 汇编程序和连接程序；还有相当完整的 Win32 的包含文件、库文件以及教程和示例等。

本书配套软件只是提供了一个最基本的 Windows 编程环境，主要是控制台输入输出等基本函数，不支持更复杂的图形窗口界面程序开发，所以本节内容基于 MASM32 开发包进行介绍。

读者可以从 Steve Hutchesson 的主页上（<http://www.movsd.com>）下载 MASM32 软件包文件（2011 年发布第 11 版）。下载下来的是一个压缩文件，解压后是一个 INSTALL.EXE 文件，可以在 Windows 2000 及以后的版本上运行，实现 MASM32 开发软件的安装。安装过程中，需要选择安装后 MASM32 所在的硬盘分区；安装程序就会将 MASM32 程序安装到所选

择分区根目录下 MASM32 目录中。安装过程要求关闭病毒检测软件。

MASM32 已经将 Windows API 常量和函数声明转换为汇编语言的包含文件，全部存放于 MASM32\INCLUDE 目录下，对应的导入库文件保存在 MASM32\LIB 目录中。MASM32 的编辑器除用于编辑源程序外，还集成了汇编、连接以及创建（Build）、调试可执行文件等功能，是一个简单的图形界面集成开发环境。

利用 MASM32 开发环境，前面例 7-1~例 7-4 的 Windows 应用程序可以删除常量定义和函数声明，但需要使用伪指令包含 WINDOWS.INC 等文件，如下所示：

```
include include\windows.inc
include include\kernel32.inc
include include\user32.inc
includelib lib\kernel32.lib
includelib lib\user32.lib
```

WINDOWS.INC 包含文件声明了所有的 Windows 数据结构和常量，例如标准输入输出句柄 STD_INPUT_HANDLE 和 STD_OUTPUT_HANDLE 常量。例题中使用的 API 函数在 KERNEL32.INC 和 USER32.INC 中声明，需要使用对应的 KERNEL32.LIB 和 USER32.LIB 导入库文件。

源程序编辑完成，取名保存。由于本书源程序的包含文件语句没有给出完整路径，所以源程序文件要保存在 MASM32 目录下。然后利用工程（Project）菜单下的控制台汇编和连接（Console Assemble & Link）命令生成 Windows 控制台可执行文件，利用汇编和连接（Assemble & Link）命令生成 Windows 图形窗口可执行文件。

【例 7-9】 一个简单的窗口应用程序。

用汇编语言创建 32 位 Windows 应用程序与使用 C++采用 API 开发没有太大区别，除了语言不同外，其程序框架、所使用的函数基本上都一样。下面程序运行后，创建一个标准的 Windows 窗口程序，包括标题栏及客户区，能够进行标准的窗口操作，例如最小化、最大化、关闭等。

如下源程序代码基于 MASM32 开发环境。请不要被这个看似庞大的源程序吓倒，因为其中大部分代码可以在任何窗口应用程序中重复使用。我们会逐步展开介绍。

```
.686
.model flat,stdcall
option casemap:none
include include\windows.inc
include include\kernel32.inc
include include\user32.inc
includelib lib\kernel32.lib
includelib lib\user32.lib

WinMain    proto :dword,:dword,:dword,:dword

.data
ClassName db "SimpleWinClass",0           ;窗口类名称
AppName    db "Win32 示例",0              ;程序名
hInstance  dd ?                            ;应用程序实例句柄
CommandLine dd ?                          ;命令行参数地址指针

.code
start:     ;调用主过程
```



```

invoke GetModuleHandle,NULL
mov hInstance,eax                ;获得实例句柄，保存
invoke GetCommandLine
mov CommandLine,eax             ;获得命令行参数地址指针，保存
invoke WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
invoke ExitProcess,eax
;WinMain 主过程
WinMain proc hInst:dword,hPrevInst:dword,CmdLine:dword,CmdShow:dword
local wc:WNDCLASSEX              ;定义窗口属性的结构变量
local msg:MSG                    ;定义消息变量
local hwnd:dword                 ;定义窗口句柄变量
;初始化窗口类变量
mov wc.cbSize,sizeof WNDCLASSEX
mov wc.style,CS_HREDRAW or CS_VREDRAW
mov wc.lfwnWndProc, offset WndProc ;WndProc 是窗口过程
mov wc.cbClsExtra,NULL
mov wc.cbWndExtra,NULL
push hInstance
pop wc.hInstance
mov wc.hbrBackground,COLOR_WINDOW+1
mov wc.lpszMenuName,NULL         ;没有使用菜单栏
mov wc.lpszClassName,offset ClassName
invoke LoadIcon,NULL,IDI_APPLICATION ;获得系统标准图标
mov wc.hIcon,eax
mov wc.hIconSm,eax
invoke LoadCursor,NULL,IDC_ARROW ;获得系统标准光标
mov wc.hCursor,eax
invoke RegisterClassEx, addr wc   ;注册窗口类
invoke CreateWindowEx,NULL,addr ClassName,addr AppName,\
WS_OVERLAPPEDWINDOW,CW_USEDEFAULT, CW_USEDEFAULT,\
CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL, hInst,NULL
mov hwnd,eax                    ;创建窗口，保存其句柄
invoke ShowWindow,hwnd,SW_SHOWNORMAL ;显示窗口
invoke UpdateWindow,hwnd         ;更新窗口
.while TRUE                      ;消息循环
    invoke GetMessage,addr msg,NULL,0,0 ;获得消息
.break .if (!eax)
; WHILE TRUE 形成无条件循环，此处当 EAX 等于 0 则跳出循环
    invoke TranslateMessage,addr msg ;翻译消息
    invoke DispatchMessage,addr msg ;分派消息
.endw
mov eax,msg.wParam
ret
WinMain endp
;窗口过程
WndProc proc hWnd:dword,uMsg:dword,wParam:dword,lParam:dword
.if uMsg==WM_DESTROY

```

```

        invoke PostQuitMessage,NULL           ;处理关闭程序的消息
    .else                                     ;不处理的消息由系统默认操作
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .endif
    xor eax,eax
    ret
WndProc    endp
end start

```

2. 主过程 WinMain

在用高级语言 C++ 开发 Windows 应用程序时，WinMain 函数是应用程序的入口点，该函数的结束也就是程序退出的出口点。汇编语言从代码段开始执行，没有 WinMain 函数。我们可以用汇编语言创建这样一个 WinMain 过程，使汇编语言更接近 C++。在调用 WinMain 前，汇编语言首先需要为这个过程准备调用参数；调用后还需要利用 WinMain 过程的返回值调用 ExitProcess 函数结束程序。

WinMain 函数的 C++ 原型是：

```

int APIENTRY WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
);

```

原型中的大写字符串表示参数类型，定义在 Windows 头文件中。这些参数类型都对应汇编语言的双字 DWORD 类型，在 WINDOWS.INC 也对这些名称进行了自定义，定义为 DWORD 类型。为了不至于对读者造成混乱，例题程序一律直接使用 DWORD 类型，后面介绍的 API 函数和 WndProc 过程也如此。所以汇编语言可以声明如下：

```

WinMain    proto :dword,:dword,:dword,:dword

```

hInstance 参数是当前应用程序的句柄实例。这个句柄可以通过使用 NULL 参数调用 GetModuleHandle 函数获得，例题程序中被保存到 hInstance 变量（API 函数的返回值通过 EAX 返回）中。hPrevInstance 参数表示前一个实例句柄，对于所有 Win32 应用程序这个参数总是 NULL。因为每个 Win32 应用程序都将创建一个独立的进程，只有一个唯一的实例，不存在前一个实例。保留该参数的目的是为了与 16 位应用程序形式上兼容。

lpCmdLine 参数用于指向命令行参数的字符串指针，这个指针可以通过调用 GetCommandLine 函数获得，例题程序中被保存在 CommandLine 变量（EAX 返回值）中。字符串按照 C++ 规范以 0 结尾。

nCmdShow 用于指定窗口的显示方式。这些方式包括 SW_SHOW（显示窗口），SW_HIDE（隐藏窗口），SW_SHOWDEFAULT（默认窗口），SW_SHOWNORMAL（正常窗口），SW_SHOWMAXIMIZED（最大化窗口），SW_SHOWMINIMIZED（最小化窗口）等。

WinMain 函数返回一个整型数值，包含在 EAX 寄存器中，可以利用这个返回值调用 ExitProcess 函数退出进程。

Windows 的 API 函数中习惯使用大写字符串表示常量，例如 SW_SHOWDEFAULT 等；其中用下画线分隔的前 2 或 3 个字母前缀表示常量所属类别，例如 SW 表示显示窗口的形式、

IDI 表示图标形式、IDC 表示光标形式、CW 表示创建窗口形式等。大多数 Windows 程序员使用匈牙利命名法为变量取名，以小写字母开头，表示变量具有的数据类型。例如 `hInstance` 的字母 `h` 表示句柄 (Handle)，`lp` 表示长型整数指针 (Long Pointer)，`n` 表示短整数 (Short)。

3. 窗口类的注册和调用

`WinMain` 函数的主要任务是：

(1) 初始化窗口类结构，对窗口类进行注册。

(2) 创建窗口、显示窗口，并更新窗口。

(3) 进入消息循环，也就是不停地检测有无消息，并把它发送给窗口进程去处理。如果为退出消息，则返回。

在 `WinMain` 函数中，首先要定义一个窗口类，并对其进行注册，即为窗口指定处理消息的过程定义光标、窗口风格、颜色等参数。“类”是面向对象程序设计中的一个最基本概念，它是用户定义的数据类型，包括数据和操作数据的函数；“对象”则是类的“实例”。

窗口属性由一个 `WNDCLASSEX` 结构设置，在 `WINDOWS.INC` 中的定义如下（注释是作者加入的）：

```
wndclassex struct
    cbSize          dd ?      ;指定该结构的大小，可以为 sizeof WNDCLASSEX
    style            dd ?      ;窗口类风格，一般为 CS_HREDRAW or CS_VREDRAW,
                                ;表示窗口高度或宽度发生变化时重新绘制窗口
    lpfnWndProc      dd ?      ;处理窗口消息的窗口过程的地址指针
    cbClsExtra       dd ?      ;分配给窗口类结构之后的额外字节数，可为 0
    cbWndExtra       dd ?      ;分配给窗口实例之后的额外字节数，可为 0
    hInstance        dd ?      ;当前应用程序的句柄实例，可以使用 WinMain 的句柄
    hIcon            dd ?      ;窗口类的图标，使用 LoadIcon 函数获得，取自系统定义的
                                ;图标或应用程序的资源
    hCursor          dd ?      ;窗口类的光标，使用 LoadCursor 函数获得，取自系统定义
                                ;的光标或应用程序的资源
    hbrBackground    dd ?      ;窗口类的背景颜色，可以使用系统定义的颜色
    lpszMenuName     dd ?      ;菜单的句柄，为 NULL 表示不显示菜单栏；或者为一个标识
                                ;菜单资源的字符串，用于显示资源定义的菜单项
    lpszClassName    dd ?      ;窗口类名称
    hIconSm          dd ?      ;图标的句柄
wndclassex ends
```

`WinMain` 函数首先用 `LOCAL` 伪指令定义了 `WNDCLASSEX` 结构的局部变量 `wc`，然后给结构成员（用英文句号“.”分隔变量名和成员名）进行赋值。对于过程中的局部变量的实质，本书在 8.2 节将进行详细介绍，此时读者按照一般变量理解即可。

Windows 操作系统维护 29 种颜色用于显示各种形状，其数值等于 0~28。例如，`COLOR_MENU` 表示菜单颜色，数值是 4；`COLOR_WINDOW` 表示窗口颜色，数值是 5；`COLOR_WINDOWFRAME` 表示窗口边框颜色，数值为 6。Windows 中，用户可以通过“控制面板”中“显示”程序的“外观”对话框改变默认显示颜色。

`LoadIcon` 函数用于加载应用程序的图标，它有两个参数。一个是应用程序实例句柄（可以用 `GetModuleHandle` 函数获得），如果设置为 `NULL`，则加载系统提供的标准图标。另一个参数是名称字符串或图标的标识符，如 `IDI_APPLICATION` 表示默认的应用程序图标，

IDI_ASTERISK 是提示性消息图标, IDI_QUESTION 表示问号图标, IDI_WINLOGO 表示 Windows 徽标。LoadIcon 函数执行结束从 EAX 返回一个图标句柄。

LoadCursor 函数用于加载应用程序的光标, 它有两个参数。一个是应用程序实例句柄(可以用 GetModuleHandle 函数获得), 如果设置为 NULL, 则加载标准光标。另一个参数是名称字符串或光标的标识符, 如 IDC_ARROW 表示标准光标(常用于对象选择), IDC_CROSS 是十字光标(常用于精确定位), IDC_HAND 是手型光标(常表示链接), IDC_HELP 是帮助光标(常说明有帮助信息)。LoadCursor 函数执行结束从 EAX 返回一个光标句柄。

有关系统颜色、图标和光标等的常量定义可以参考 WINDOWS.INC 文件。

完成窗口类属性的设置, 就可以调用 RegisterClassEx 函数进行窗口类注册了, 它有一个参数需要指向 WNDCLASSEX 结构变量 wc。

4. 窗口的创建、显示和更新

一旦完成窗口注册, 接着就是为应用程序创建一个实际的窗口。创建窗口需要调用 CreateWindowEx 函数。在 Win32 程序员手册中, 它的原型如下:

```
HWND CreateWindowEx(  
    DWORD dwExStyle,           // extended window style  
    LPCTSTR lpClassName,       // pointer to registered class name  
    LPCTSTR lpWindowName,      // pointer to window name  
    DWORD dwStyle,             // window style  
    int x,                     // horizontal position of window  
    int y,                     // vertical position of window  
    int nWidth,                // window width  
    int nHeight,               // window height  
    HWND hWndParent,           // handle to parent or owner window  
    HMENU hMenu,               // handle to menu, or child-window identifier  
    HINSTANCE hInstance,       // handle to application instance  
    LPVOID lpParam              // pointer to window-creation data  
);
```

参数 dwExStyle 是窗口的扩展风格, NULL 表示不使用。参数 dwStyle 是窗口风格。WS_OVERLAPPEDWINDOW 表示创建一个标准 Windows 窗口, 包括有标题栏、系统菜单按钮、粗边框以及最小化、最大化和关闭按钮。它实际上包括了 WS_OVERLAPPED(标题栏和边框)、WS_CAPTION(标题栏)、WS_SYSMENU(系统菜单按钮)、WS_THICKFRAME(粗边框)、WS_MINIMIZEBOX(最小化按钮)和 WS_MAXIMIZEBOX(最大化按钮)风格, 与 WS_TILEDWINDOW 风格一样。

lpClassName 参数是注册的窗口类名称指针。lpWindowName 是程序名指针。参数 x, y 和 nWidth, nHeight 依次是窗口的水平、垂直位置和宽度、高度; 可以是具体的数值, 使用 CW_USEDEFAULT 表示使用系统默认值。hWndParent 是父窗口句柄, hMenu 是菜单句柄或子窗口标识符, hInstance 是应用程序实例句柄。lpParam 参数是指向窗口数据的指针。

CreateWindowEx 函数创建窗口成功, 在 EAX 返回其句柄, 否则返回 NULL。窗口创建后并不会马上显示, 需要 ShowWindow 函数显示窗口; 还需要调用 UpdateWindow 函数对窗口更新。显示窗口和更新窗口函数都需要使用窗口的实例句柄, 显示窗口时还要指明显示方式, 常量 SW_SHOWNORMAL 表示正常窗口。

5. 消息循环

屏幕上有了显示窗口，程序就必须准备好接收用户的键盘和鼠标输入。Windows 操作系统为每个 Windows 程序维持一个消息队列（Message Queue）。当一个输入事件发生时，Windows 操作系统翻译该事件成为一个消息，并放置于消息队列中。本例程序使用一个循环控制伪指令实现一个 WHILE 循环结构，从消息队列中检索消息、翻译消息和分派消息，即消息循环。

消息用 MSG 结构表达，在 MASM32 软件包的 WINDOWS.INC 文件中，类型声明如下：

```
msg struct
    hwnd    dd ?
    message dd ?
    wParam  dd ?
    lParam  dd ?
    time    dd ?
    pt      point <>
msg ends
```

参数 hwnd 指示窗口，其窗口过程接收消息。message 参数是一个消息编号，说明消息的类型。参数 wParam 和 lParam 指明消息的附加信息，它们的具体含义取决于消息类型。time 参数标明消息发送的时间。pt 参数是 POINT 数据类型，它又是一个结构类型，指示当消息发送时的光标位置。在 WinMain 过程开始，定义有一个 MSG 结构类型的 msg 变量。

在消息循环的开始，GetMessage 函数从消息队列检索一个消息。该函数的第 1 个参数是指向 msg 消息变量的指针。第 2 个参数是要接收消息的窗口句柄，如果为 NULL 有特别含义，表示程序需要其所有窗口的消息。第 3 个和第 4 个参数都是整数值，分别表示接收的最低和最高消息编号；如果它们都是 0，则表示 GetMessage 函数将返回所有消息。

消息变量的值由操作系统根据用户的输入设置。当收到除 WM_QUIT 之外的消息时，GetMessage 函数返回非零数值，即逻辑真 TRUE。而当收到 WM_QUIT 消息时，GetMessage 函数返回零，即逻辑假 FALSE，此时语句“`.BREAK .IF (!eax)`”使程序跳出消息循环，WinMain 函数返回。API 函数用 EAX 返回值，所以 msg.wParam 赋值给 EAX，用 RET 指令返回主程序。若存在错误，GetMessage 函数返回-1。

如果进程需要从键盘接收字符输入，消息循环中必须包括 TranslateMessage 函数。每次用户按键，Windows 生成一个虚拟键消息，它是一个虚拟键代码而不是字符代码值。为了得到这个字符代码值，消息循环需要使用 TranslateMessage 函数将虚拟键消息翻译成字符消息，并放回到应用程序的消息队列。然后消息循环利用 DispatchMessage 函数将消息分派给窗口过程，也就是在注册窗口时的 WndProc 过程。

6. 窗口过程

前面介绍的 WinMain 过程应该说只是辅助操作，真正的动作处理是在窗口过程中，本例程序取名为 WndProc 过程。窗口过程决定了在客户区的显示内容，以及程序如何响应用户输入。WndProc 过程通常如下定义：

```
WndProc proc hwnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
```

其中冒号后的大写字符串表示参数类型，其实在 WINDOWS.INC 文件中被重新定义为 DWORD 类型，所以都可以直接写作 DWORD。它的 4 个参数含义与 MSG 结构的前 4 个结

构字段一样，依次是窗口句柄、消息编号和 2 个消息附加信息。其中参数 `uMsg` 是一个数值，表示消息类型。包含文件中以 `WM` 前缀（Window Message）的标识符定义了 Windows 的各种消息，例如：

- ⊙ `WM_LBUTTONDOWN`——按下鼠标左键产生的消息。
- ⊙ `WM_RBUTTONDOWN`——按下鼠标右键产生的消息。
- ⊙ `WM_CLOSE`——主窗口关闭时产生的消息。
- ⊙ `WM_DESTROY`——用户结束程序运行时产生的消息。

窗口过程根据参数 `uMsg` 得到消息，并转到不同的分支去处理。窗口过程处理的消息，返回 Windows 时要在 `EAX` 中赋值 0；不处理的消息，必须调用 `DefWindowProc` 函数由操作系统按照默认方式进行处理，并将其返回值返回给窗口过程，以确保发送给应用程序的每条消息都能够得到响应。调用 `DefWindowProc` 函数需要使用与窗口过程相同的参数。

本例程序的窗口过程仅处理了 `WM_DESTROY` 消息，即使用调用 `PostQuitMessage` 函数的标准方式。`PostQuitMessage` 函数向消息队列发送 `WM_QUIT` 消息，并立即返回。该函数带有一个退出代码参数，作为 `WM_QUIT` 消息的 `wParam` 参数。

至此，我们从头到尾分析了 Windows 窗口应用程序。不过，这个程序生成的标准窗口没有任何功能。现在，增加单击鼠标左键弹出消息窗口的功能，实现如图 7-3 所示的运行效果。

【例 7-10】 弹出消息的窗口应用程序。

我们只需要在例 7-9 源程序的基础上进行简单增添。

数据段增加一个字符串：

```
szText          db '欢迎进入 32 位 Windows 世界! ',0
```

窗口过程中增加两条语句：

```
invoke PostQuitMessage,NULL      ;处理关闭程序的消息
.elseif uMsg==WM_LBUTTONDOWN     ;处理单击鼠标左键的消息
    invoke MessageBox,NULL,addr szText, addr AppName,MB_OK
.else                             ;不处理的消息由系统默认操作
```



图 7-3 例 7-10 弹出消息的窗口应用程序

通过上述若干例题程序的分析和实践，相信读者对使用汇编语言编写 32 位 Windows 应用程序有了初步理解或掌握。我们看到，只要充分利用 MASM 的高级语言程序设计特性，尤其是 `PROTO` 和 `INVOKE` 伪指令，调用 Windows API 函数，同时借助 Htch 提供的免费 MASM32 软件，完全可以采用汇编语言开发 32 位 Windows 应用程序。但是，我们也看到，这是一个相对烦琐的过程。即使是采用 C++调用 API 函数编写 Window 应用程序，也将涉及非常繁杂的技术细节，许多 C++程序员往往对设备句柄、消息机制、字体度量、位图和映射方式等搞得一头雾水。另外，限于本书的目的，许多内容不能展开，所以读者未必能够完全

理解其中的技术细节，有兴趣的读者可以进一步阅读相关文献或者深入学习 MASM32 提供的帮助、教程、示例等内容。

习 题 7

7.1 简答题

- (1) 什么是应用程序接口 API?
- (2) 什么是静态连接?
- (3) 运行 Windows 应用程序，有时为什么会提示某个 DLL 文件不存在?
- (4) ADDR 与 OFFSET 有何不同?
- (5) ExitProcess 函数可以按汇编语言习惯全部使用小写字母表示吗?
- (6) Win32 API 中可以使用哪两种字符集?
- (7) 为什么调用 API 函数之后，ECX 等寄存器改变了?
- (8) 条件控制“.IF”伪指令的条件是在汇编阶段进行判断吗?
- (9) 为什么 32 位 API 函数的地址指针也可以转换为汇编语言的双字类型?
- (10) 在 MASM32 软件包支持下的汇编语言程序中为什么没有看到对 Windows 常量、函数等的定义和声明呢?

7.2 判断题

- (1) Windows 可执行文件中包含动态连接库中的代码。
- (2) 导入库文件和静态子程序库文件的扩展名都是 LIB，所以两者性质相同。
- (3) INVOKE 语句只能传递主存操作数，不能传递寄存器值。
- (4) Windows 控制台是命令行窗口，也就是 MS-DOS 窗口。
- (5) 类似高级语言，汇编语言的使用结构变量也需要先说明结构类型。
- (6) PROC 伪指令可使用 USES 操作符，但 PROTO 伪指令不可以使用。
- (7) 在宏定义中，LOCAL 伪指令声明标识符；而在过程定义中，LOCAL 伪指令用于分配局部变量。
- (8) 条件汇编 IF 和条件控制 IF 伪指令都包括条件表达式，它们的表达形式一样。
- (9) 条件控制 IF 令和循环控制 WHILE 伪指令中的条件表达式具有相同的表达形式。
- (10) MASM32 软件包只支持 32 位图形界面应用程序的开发，不支持控制台应用程序开发。

7.3 填空题

- (1) Windows 系统有 3 个最重要的系统动态连接库文件，它们是_____、_____和_____。
- (2) 进行 Windows 应用程序开发时，需要_____库文件；执行该应用程序时则需要对应的_____库文件。
- (3) 获得句柄函数 GetStdHandle 执行结束，使用_____提供返回结果。
- (4) 函数 GetStdHandle 需要一个参数，对标准输入设备应该填入_____数值，对标准输出设备应该填入_____数值，对标准错误设备应该填入_____数值。
- (5) 调用 ReadConsole 函数时，用户在键盘上按下数字 8，然后回车，则键盘缓冲区的内容依次是_____。

(6) WriteConsole 和 ReadConsole 函数的参数类似, 都有 5 个, 第 1 个参数是_____, 第 2 个参数是输出或输入缓冲区的_____, 第 3 个参数是输出或输入的字符_____, 第 4 个参数指向实际输出或输入字符个数的变量, 第 5 个参数一般要求代入_____。

(7) 消息窗口 MessageBox 函数有 4 个参数, 第一个是 0, 第 2 个是要显示字符串的_____, 第 3 个是_____的地址指针, 第 4 个参数指明窗口形式。注意字符串要使用_____作为结尾标志。

(8) 使用获取系统时间函数 GetLocalTime 需要定义一个_____结构变量, 其中返回系统时间数值, 这些数值采用二进制编码, 例如日期返回的编码是: 0019H, 它表示日期是_____日。

(9) 使用扩展的 PROC 伪指令编写子程序比较方便, 例如子程序中需要保护和恢复 ESI 和 EDI, 就只需要使用_____就可以了。

(10) MASM 进行汇编时生成最大化源代码列表, 其中语句前使用字母_____表示是通过包含文件插入的语句, 使用“*”符号的语句常是_____的代码, 而语句前的数字则说明是_____语句。

7.4 执行第 6 章介绍的 CPUID 指令, 直接使用控制台输出函数将处理器识别字符串显示出来。

7.5 直接使用控制台输入和输出函数实现例 7-2 的功能(不使用 READMSG 和 DISPMSG 子程序形式)。注意, 输入或输出句柄只要各获得一个即可。

7.6 直接使用控制台输出函数实现某个主存区域内容的显示(见习题 5.13)。要求改进显示形式, 例如每行显示 16 个字节(128 位), 每行开始先显示首个主存单元的偏移地址, 然后用冒号分隔主存内容。

7.7 执行第 6 章介绍的 CPUID 指令, 在消息窗口显示处理器识别字符串, 要求该消息窗有“OK”和“Cancel”两个按钮。

7.8 参考例 5-10, 利用 MessageBox 函数创建的消息窗口显示 32 位通用寄存器内容。

7.9 利用获得系统时间函数, 将年月日时分秒星期等时间完整的显示出来。可以创建一个控制台程序, 也可以创建一个消息窗口程序。

7.10 结构数据类型如何说明、结构变量如何定义、结构字段如何引用?

7.11 条件控制伪指令的条件表达式中, 逻辑与“&&”表示两者都为真, 整个条件才为真; 逻辑或“||”表示两者之一为真, 整个条件就为真。对如下两个程序段(VAR 是一个双字变量):

(1) 逻辑与条件

```
.if (var==5) && (eax!=ebx)
    inc eax
.endif
```

(2) 逻辑或条件

```
.if (var==5) || (eax!=ebx)
    dec ebx
.endif
```

请直接使用处理器指令实现上述分支结构, 并比较汇编程序生成的代码序列。

7.12 对于如下两个程序段：

(1) WHILE 循环结构

```
.while eax!=10
    mov [ebx*4],eax
    inc eax
.endw
```

(2) UNTIL 循环结构

```
.repeat
    mov [ebx*4],eax
    inc eax
.until eax==10
```

请直接使用处理器指令实现上述循环结构，并比较汇编程序生成的代码序列。

7.13 使用条件控制.IF 伪指令编写习题 4.14 程序，并生成完整的列表文件。

7.14 用条件控制.IF 和循环控制.WHILE 伪指令编写习题 4.19 程序，并生成完整的列表文件。

7.15 调用 GetCommandLine 函数，可以从 EAX 返回指向命令行输入字符串（包含路径、文件名和参数）。现要求编程，利用 MessageBox 函数输出这个字符串。

7.16 在 Windows 窗口应用程序例 7-10 基础上，增加单击鼠标右键弹出另一个消息窗口的功能，在 MASM32 开发环境生成可执行文件。

第 8 章 与 Visual C++ 的混合编程

用汇编语言开发的程序虽然有占用存储空间小、运行速度快、能直接控制硬件等优点，但它与机器密切相关、移植性差，而且编程烦琐、对汇编语言程序员要求较高。所以，软件开发通常采用高级语言，以提高开发效率，但某些部分，如程序的关键部分、运行次数很多的部分、运行速度要求很高的部分、直接访问硬件的部分等，则利用汇编语言编写，以提高程序的运行效率。汇编语言与高级语言或不同的高级语言间，通过相互调用、参数传递、共享数据结构和数据信息而形成程序的过程就是混合编程。

汇编语言与 C、C++ 语言有两种混合编程方法：嵌入汇编和模块连接。本书以 MASM 汇编语言和 Visual C++ 6.0 为例，基于 32 位 Windows 控制台进行介绍。

各种程序设计语言都有自己的语法规则和格式规范，当使用不同语言进行混合编程时，就需要按照相互都能够支持的规则和规范编写各自的程序部分，然后形成一个完整的程序。所以，学习混合编程其实就是了解和使用这些规则和规范，只是方法问题，谈不上技术或者原理。本章重点介绍局部变量和堆栈帧的工作原理。虽然这个问题出现在高级语言中，却适合通过汇编语言来讲解清楚。

8.1 嵌入汇编

嵌入汇编是指直接在 C、C++ 语言的源程序中插入汇编语言指令，也称为内嵌汇编、内联汇编或行内（in-line）汇编。

Visual C++ 使用 “`__asm`” 关键字（注意，`asm` 前是两个下画线、中间并没有空格；但 Visual C++ 6.0 也支持一个下画线的格式 `_asm`，目的是与以前版本保持兼容）指示嵌入汇编，不需要独立的汇编程序就可以进行正常编译和连接。

Visual C++ 嵌入汇编最好采用花括号的汇编语言程序片段形式，例如：

```
// __asm 程序片段
__asm
{
    mov eax,01h          //支持汇编语言的注释格式
    mov dx,0xD007        ;0xD007=0D007H，支持 C、C++的数据表达形式
    out dx,eax
}
```

Visual C++ 也支持单条汇编语言指令形式，例如：

```
//单条 __asm 汇编指令形式
__asm mov eax,01h
__asm mov dx,0D007h
__asm out dx,eax
```

另外，可以使用空格在一行分隔多个 `__asm` 汇编语言指令，例如：

```
//多个 __asm 语句在同一行时，用空格将它们分开
__asm mov eax,01h __asm mov dx,0xD007 __asm out dx,eax
```

上面 3 种形式产生相同的代码，但第一种形式具有更多的优点，因为可以将 C++ 代码与汇编代码明确分开，避免混淆。如果将 `__asm` 指令和 C++ 语句放在同一行且不使用括号，编译器就分不清汇编代码到什么地方结束和 C++ 从哪里开始。`__asm` 花括号中的程序片段不影响变量的作用范围。`__asm` 块允许嵌套，嵌套也不影响变量的作用范围。

1. 嵌入汇编语句中使用汇编语言的注意事项

① Visual C++ 6.0 支持通用整数和浮点指令集，以及 MMX 指令集的嵌入汇编。对于还不能支持的指令，Visual C++ 提供了 `_emit` 伪指令进行扩展。

`_emit` 伪指令类似 MASM 中的 DB 变量定义伪指令，可以用来定义一字节的内容，并且只能用于程序代码；例如：

```
//定义汇编指令代码的宏，这 2 字节内容是 CPUID 指令的机器代码
#define cpu-id __asm _emit 0x0F __asm _emit 0xA2
//使用 C++ 的宏
__asm{ cpu-id }
```

② 嵌入汇编可以使用 MASM 的表达式，这个表达式是操作数和操作符的组合，产生一个数值或地址。嵌入汇编还可以使用 MASM 的注释风格。

③ 嵌入汇编虽然可以使用 C++ 的数据类型和数据对象，但不可以使用 MASM 的绝大多数伪指令和宏汇编方法。

例如，不能使用 DB、DW、DD 等伪指令和 DUP 等操作符，也不能使用 MASM 的结构 STRUCT 和记录 RECORD 等，不支持 MASM 的宏伪指令（如 MACRO、ENDM、REPEAT、FOR、FORC 等）和宏操作符（如 !、&、%等）。

嵌入汇编不支持 MASM 6.0 引入的 LENGTHOF 和 SIZEOF 操作符，但可以使用 LENGTH、SIZE、TYPE 操作符来获取 C++ 变量和类型的大小。LENGTH 用来返回数组元素的个数，对非数组变量返回值为 1；TYPE 返回 C++ 类型或变量的大小，如果变量是一个数组，它返回数组单个元素的大小。SIZE 返回 C++ 变量的大小，即 LENGTH 和 TYPE 的乘积。

例如，对于数组 `int iarray[8]`（int 类型是 32 位，4 字节），则：

LENGTH `iarray` 返回 8（等同于 C++ 的 `sizeof(iarray)/sizeof(iarray[0])`），表示数组有 8 个元素。

TYPE `iarray` 返回 4（等同于 C++ 的 `sizeof(iarray[0])`），表示每个数组元素占 4 字节。

SIZE `iarray` 返回 32（等同于 C++ 的 `sizeof(iarray)`），表示数组占 32 字节。

嵌入汇编中不能使用 OFFSET，但可以使用 LEA 指令获得偏移地址。

嵌入汇编语句中可以使用 PTR 指明操作数类型，例如：

```
__asm inc byte ptr [esi]
```

④ 在用嵌入汇编书写的函数中，不必保存 EAX、EBX、ECX、EDX、ESI 和 EDI 寄存器，但必须保存函数中使用的其他寄存器（如 DS、SS、ESP、EBP 和整数标志寄存器）。例如，用 STD 和 CLD 改变方向标志位，就必须保存标志寄存器的值。

嵌入汇编引用段时应该通过寄存器而不是通过段名；段超越时，必须清晰地用段寄存器说明，如 `ES:[EBX]`。

2. 嵌入汇编语句中使用 C++ 语言的注意事项

① 嵌入汇编可使用 C++ 的下列元素：符号（包括标号、变量、函数名）、常量（包括符

号常量、枚举成员)、宏和预处理指令、注释 (/* */和//, 也可以使用汇编语言的注释风格)、类型名及结构、联合的成员。

嵌入汇编语句使用 C++ 符号也有一些限制: 每条汇编语言语句只包含一个 C++ 符号 (包含多个符号只能通过使用 LENGTH、TYPE 和 SIZE 表达式); 引用函数前必须在说明其原型 (否则编译程序将分不出是函数名还是标号); 不能使用与 MASM 保留字相同的 C++ 符号 (如指令助记符和寄存器名), 也不能识别结构 Structure 和联合 union 关键字。

② 嵌入汇编语句中, 可以使用汇编语言格式表示整数常量 (如 378H), 也可以采用 C++ 的格式 (如 0x378)。

③ 嵌入汇编语句不能使用 C++ 的专用操作符, 如 << ; 对两种语言都有的操作符在汇编语句中作为汇编语言操作符, 如 * 或 []。例如:

```
int array[6];           //C++语句中, []表示数组的某个元素
__asm mov array[6],ebx  //汇编语言中, []表示距离标识符的字节偏移量
```

④ 嵌入汇编中可以引用包含该 __asm 作用范围内的任何符号 (包括变量名), 通过使用变量名引用 C++ 的变量。例如, 若 var 是 C++ 中的整型 int 变量, 则可以使用如下语句:

```
__asm mov eax,var
```

如果类、结构、联合的成员名字唯一, __asm 中可不说明变量或类型名就可以引用成员名, 否则必须说明。例如:

```
struct first_type
{
    char *carray;
    int same_name;
};
struct second_type
{
    int ivar;
    long same_name;
};
struct first_type ftype;
struct second_type stype;
__asm
{
    mov ebx,offset ftype
    mov ecx,[ebx]ftype.same_name  //必须使用 ftype
    mov esi,[ebx].carray          //可以不使用 ftype (也可以使用)
}
```

⑤ 利用 C、C++ 的宏可以方便地将汇编语言代码插入源程序中。C、C++ 宏将扩展成为一个逻辑行, 所以书写具有嵌入汇编的 C、C++ 宏时, 应遵循下列规则: 将 __asm 程序段放在括号中, 每条汇编语言指令前必须有 __asm 标志, 应该使用 C 语言的注释风格 (/* */), 不要使用 C++ 语言的单行注释 (//) 和汇编语言的分号注释 (;) 方式。例如:

```
#define PORTIO __asm \
/* Port output */ \
{ \
    __asm mov eax,01h \
    __asm mov dx,0xD007 \
```

```

    __asm out dx,eax    \
}

```

该宏展开为一个逻辑行（上面的“\”是续行符）：

```

__asm /* Port output */ { __asm mov eax,01h __asm mov dx,0xD007 __asm out dx,eax}

```

⑥ 嵌入汇编中的标号和 C++ 的标号相似，它的作用范围为定义它的函数中。汇编转移指令和 C++ 的 goto 语句都可以跳转到 __asm 块内或块外的标号。

__asm 块中定义的标号对大小写不敏感，汇编语言指令跳转到 C++ 语言中的标号也不分大小写，C++ 语言中的标号只有使用 goto 语句时对大小写敏感。

【例 8-1】 嵌入汇编计算数组平均值函数。

对于计算 32 位有符号数平均值的例 7-5（类似 16 位版本的例 5-7）使用 C++ 语言编写主程序，求平均值 MEAN 函数使用嵌入汇编。注意，本程序都基于 32 位 Windows 平台。

```

#include <iostream.h>
#define COUNT 10
long mean(long d[], long num);
int main()
{
    long array[COUNT] = {675, 354, -34, 198, 267, 0, 9, 2371, -67, 4257};
    cout<<"The mean is \t"<<mean(array,COUNT)<<endl;
    return 0;
}
long mean(long d[], long num)
{
    long temp;           // 定义局部变量，用于返回值
    __asm {              // 嵌入式汇编代码部分（参考例 7-5 程序的 32 位版）
        mov ebx,d         ;EBX=数组地址
        mov ecx,num       ;ECX=数据个数
        xor eax,eax       ;EAX 保存和值
        xor edx,edx       ;EDX=指向数组元素
mean1:
        add eax,[ebx+edx*4] ;求和
        add edx,1         ;指向下一个数据
        cmp edx,ecx       ;比较个数
        jb mean1          ;循环
        cdq               ;将累加和 EAX 符号扩展到 EDX
        idiv ecx          ;有符号数除法，EAX=平均值（余数在 EDX 中）
        mov temp,eax
    }
    return(temp);
}

```

函数的局部变量要在嵌入汇编模块之外声明。

在 Visual C++ 集成开发环境中，建立一个 Win32 控制台程序的项目，创建上述源程序后加入该项目。然后，进行编译连接就产生一个可执行文件（如果不熟悉 Visual C++ 开发环境，可以参考 8.4 节）。

8.2 模块连接

模块连接方式是不同编程语言之间混合编程经常使用的方法。各种语言的程序分别编写，利用各自的开发环境编译形成目标代码（OBJ）模块文件，然后将它们连接在一起，最终生成可执行文件。

相对来说，Visual C++直接支持嵌入汇编方式，不需要独立的汇编系统和另外的连接步骤。所以，嵌入汇编比模块连接方式更简单方便。但是嵌入汇编的主要缺点是缺乏可移植性。例如，运行于 IA-32 处理器的嵌入汇编代码不能移植到其他非兼容的处理器上，然而模块连接方式却可以比较方便地为不同处理器平台提供不同的外部目标代码模块。

8.2.1 约定规则

进行模块连接，必须对它们的接口、参数传递、返回值及寄存器的使用、变量的引用等作出约定，以保证连接程序能得到必要的信息，进行正确的连接。

1. 采用一致的调用规范

C、C++语言与汇编语言混合编程的参数传递通常利用堆栈，调用规范（Calling Convention）决定利用堆栈的方法和命名约定，两者要一致。

Visual C++具有 3 种调用规范：`_cdecl`、`_stdcall` 和 `_fastcall`，默认采用 `_cdecl`（`_cdecl`，即 `c declare`，也就是 C 语言调用规范）调用规范，Windows 的 API 函数等采用 `_stdcall` 调用规范。

MASM 汇编语言利用“语言类型”（Language Type）确定调用规范和命名约定。例如，使用 Visual C++的 `_cdecl` 调用规范要对应 MASM 的 C 语言类型，使用 Visual C++的 `_stdcall` 调用规范要对应 MASM 的 `STDCALL` 语言类型。

2. 声明共用函数和变量

C++语言和汇编语言的共用过程名、变量名需要进行声明，并且标识符要求一样。注意，C++语言对标识符区别字母的大小写，而汇编语言不区分大小写（为避免混乱，可以按照高级语言的标识符形式书写）。

在 C++程序中，采用 `extern "C" { }` 对所要调用的外部过程、函数、变量予以说明，说明形式如下：

```
extern "C" { 返回值类型 调用规范 函数名称(参数类型表); }
extern "C" { 变量类型 变量名; }
```

汇编程序中供外部使用的标识符应具有 `PUBLIC` 属性，使用外部标识符要利用 `EXTERN` 声明。MASM 中过程名默认具有 `PUBLIC` 属性，也可以利用 `PUBLIC` 伪指令或者 `PROTO` 伪指令说明。

3. 入口参数和返回参数的约定

C、C++语言中不论采用何种调用规范，传送的参数形式都是“传值”（by Value），但数组除外（因为数组名表示的是第一个元素的地址）。参数“传址”（by Reference）应利用指针数据类型。

Visual C++的 char、short 和 long (包括 int) 的数据类型依次是字节、字和双字, 与 MASM 数据类型对应关系如表 8-1 所示。但不论何种整数类型, 进行参数传递时, 都扩展成 32 位。注意, 32 位 Visual C++版本中整型 int 是 4 字节, 函数调用使用 32 位偏移地址, 所有的地址参数都是 32 位偏移地址, 在堆栈中占 4 字节。

表 8-1 Visual C++与 MASM 数据类型对应关系

Visual C++的数据类型	MASM 的数据类型	Visual C++的数据类型	MASM 的数据类型	字节数
unsigned char	BYTE	char	SBYTE	1
unsigned short	WORD	short	SWORD	2
unsigned int	DWORD	int	SDWORD	4
unsigned long	DWORD	long	SDWORD	4

Visual C++函数返回参数时, 8 位值 (如 char 或 bool 类型) 在 AL 返回, 16 位值 (如 short 类型) 在 AX 返回, 32 位值 (如 long、int 或地址指针) 存放在 EAX 寄存器中返回; 64 位返回值存放在 EDX (高 32 位) 和 EAX (低 32 位) 寄存器对中; 更大数据则将它们地址指针存放在 EAX 中返回。

高级语言使用堆栈传递参数, 从方便混合编程的角度, 汇编语言的模块可以使用扩展过程定义 PROC 伪指令。

【例 8-2】 模块连接计算数组平均值函数。

对于例 8-1 求平均值 MEAN 函数, 可使用汇编语言单独编写成一个模块。原例 8-1 需要删除函数定义, 同时将函数声明修改为:

```
extern "C" { long mean(long d[], long num); }
```

汇编语言过程的源程序文件如下:

```
.686
.model flat,c
mean    proto d:ptr dword,num:dword    ;过程声明
.code
mean    proc uses ebx ecx edx,d:ptr dword,num:dword    ; 过程定义
        mov ebx,d                                ;EBX=数组地址
        mov ecx,num                              ;ECX=数据个数
        .....                                  ;同例 8-1 (参考例 7-5 的 32 位版), 略
        idiv ecx                                ;有符号数除法, EAX=平均值 (余数在 EDX 中)
        ret
mean    endp
end
```

先对上述汇编语言程序进行汇编, 生成目标模块文件 (使用 COFF 格式)。然后将该模块文件加入到 Visual C++的 Win32 控制台程序的项目中。最后, 同 C++源程序一起编译连接创建可执行文件。

本例程序中, 汇编语言使用 C 语言类型, C++程序默认采用_cdecl, 两者保持一致。如果汇编语言使用 STDCALL 语言类型, 则 C++程序必须在函数声明语句中明确指示采用_stdcall 规范。

8.2.2 堆栈帧

堆栈在混合编程中起着至关重要的作用。堆栈在过程调用中为传递参数、返回地址、局部变量和保护寄存器所保留的堆栈空间，被称为堆栈帧（Stack Frame），其创建步骤如下：

- (1) 主程序把传递的参数压入堆栈。
- (2) 调用子程序时，返回地址压入堆栈。
- (3) 子程序中，EBP 压入堆栈；设置 EBP 等于 ESP，通过 EBP 访问参数和局部变量。
- (4) 子程序有局部变量，ESP 减去一个数值，在堆栈预留局部变量使用的空间。
- (5) 子程序要保护的寄存器压入堆栈。

建议大家回顾求平均值例 5-7（及例 7-5）程序所创建的堆栈帧，下面则引出局部变量。

1. 局部变量

汇编语言习惯使用寄存器进行编程，常用寄存器作为临时变量来替代局部变量。为了理解局部变量，可利用扩展过程定义支持局部变量的特性，增加一个局部变量（Local Variable）来修改例 8-2 的求平均值程序。

【例 8-3】 使用局部变量求有符号数的平均值程序。

请把注意力集中在局部变量 SUM 的使用上（语句注释用 “**” 标示）：

```
.686
.model flat,c
mean    proto d:ptr dword,num:dword    ;过程声明
.code
mean    proc c uses ebx ecx edx,d:ptr dword,num:dword
        local sum:dword                ;** 定义局部变量
        mov ebx,d                      ;EBX=数组指针
        mov ecx,num                    ;ECX=元素个数
        mov sum,0                      ;** SUM 保存和值
        xor edx,edx                    ;EDX=指向数组元素
mean1:   mov eax,[ebx+edx*4]             ;取一个数据
        add sum,eax                    ;** 求和
        add edx,1                      ;指向下一个数据
        cmp edx,ecx                    ;比较个数
        jb mean1                      ;循环
        mov eax,sum                    ;** 取和值
        cdq                           ;将累加和 EAX 符号扩展到 EDX
        idiv ecx                       ;有符号数除法，EAX=平均值（余数在 EDX 中）
        ret
mean     endp
end
```

使用如下汇编命令：

```
ml /c /coff /fI /Sa eg803.asm
```

这样将生成完整的列表文件（其中前面有 “*” 的指令由汇编程序生成）：

```
mean    proc c uses ebx ecx edx,d:ptr dword,num:dword
        local sum:dword                ;** 定义局部变量
00000000 55          *      push  ebp
```



```

00000001  8B EC      *      mov    ebp, esp
00000003  83 C4 FC      *      add    esp, 0FFFFFFFCh
00000006  53          *      push   ebx
00000007  51          *      push   ecx
00000008  52          *      push   edx
00000009  8B 5D 08      mov    ebx,d          ;EBX=数组指针
                        .....      ;同源程序, 略
                        ret
0000002B  5A          *      pop    edx
0000002C  59          *      pop    ecx
0000002D  5B          *      pop    ebx
0000002E  C9          *      leave
0000002F  C3          *      ret    00000h
00000030      mean endp

```

通过列表文件看到, 指令“add esp,0FFFFFFFCh”使得 ESP 减 4 (32 位补码 FFFFFFFCh 的真值是-4, 该指令自动进行符号扩展与 ESP 相加), 为双字局部变量 SUM 预留了 4 字节。通过列表文件的符号部分还可以看到, 子程序通过“EBP-4”访问局部变量 SUM, 通过“EBP+8”访问参数 D, 通过“EBP+0Ch”访问参数 NUM。

最后有条 LEAVE 指令的功能相当于如下两条指令 (参见后面详解):

```

mov esp,ebp
pop ebp

```

所以, 也许大家这时才体会到高级语言对局部变量的说明:

局部变量是在程序运行时通过堆栈创建的, 只有该过程内的语句可以访问这个局部变量, 过程执行结束局部变量随之消失。

2. 高级语言的堆栈帧

实际上, MASM 创建的局部变量与高级语言的局部变量一样, 通过如下 C++程序求平均值程序进行对比。

【例 8-4】 计算数组平均值的 C++函数。

```

..... //同例 8-1, 省略
long mean(long d[], long num)
{
    long i,temp=0;
    for(i=0;i<num;i++) temp=temp+d[i];
    temp=temp/num;
    return(temp);
}

```

在 Visual C++集成开发环境下创建可执行文件, 注意在项目配置中 C/C++标签的列表文件类型 (Listing file type) 中选择含有汇编代码的选项 (建议选择 Assembly with Source Code), 生成含汇编语言的列表文件 (含有源代码和汇编语言代码, 扩展名是.ASM)。如果选项是 Assembly、Machine Code 和 Source, 列表文件还包括机器代码, 但生成的列表文件的扩展名是.COD。如果选项是 Assembly-Only Listing, 仅生成汇编语言的列表文件 (扩展名是.ASM)。也可以使用 DUMPBIN 反汇编可执行文件, 或者通过 OBJ 文件获得汇编语言代码。

要完全读懂 Visual C++生成的汇编语言列表文件内容, 还需要补充 MASM 知识和有关编

译技术，下面主要针对求平均值函数展开讨论，并注意其堆栈帧，如图 8-1 所示。

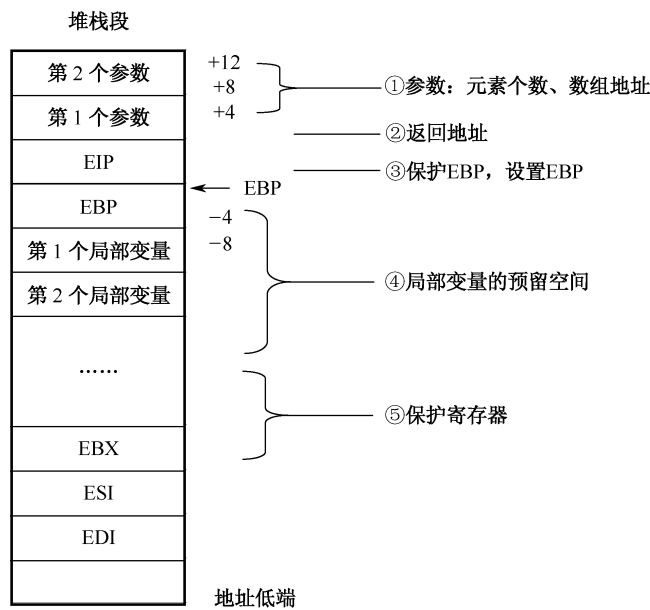


图 8-1 求平均值函数的堆栈帧

首先采用 Visual C++默认的调试（Debug）版本进行开发、生成含汇编语言的列表文件。主函数中调用 MEAN 函数的汇编代码（包括注释）如下：

```
push 10 ; 0000000aH
lea eax, DWORD PTR _array$[ebp]
push eax
call ?mean@@YAJQAJJ@Z ; mean
add esp, 8
```

第 1 条 PUSH 指令将第 1 个参数（10，即数组元素个数）压入堆栈。

第 2 条使用 LEA 指令获得数组（array）的地址。主函数同样使用堆栈区域安排数组，位置由 “_array\$[ebp]” 指示，所以 LEA 获得其有效地址（不能使用 OFFSET 操作符）。

第 3 条 PUSH 指令压入堆栈第 2 个参数（数组地址）。

第 4 条 CALL 指令调用函数 MEAN，不过函数名被编译程序进行了修饰（详见后面调用规范的说明）。

第 5 条 ADD 指令增量 ESP，调用程序平衡堆栈。

求平均值 MEAN 函数的汇编代码（包括注释）如下：

```
_d$ = 8
_num$ = 12
_i$ = -4
_temp$ = -8
?mean@@YAJQAJJ@Z PROC NEAR ; mean, COMDAT

; 11 : {

push ebp
```

```

        mov     ebp, esp
        sub     esp, 72          ; 00000048H
        push    ebx
        push    esi
        push    edi
        lea     edi, DWORD PTR [ebp-72]
        mov     ecx, 18          ; 00000012H
        mov     eax, -858993460  ; ccccccccH
        rep stosd

; 12   :   long i,temp=0;

        mov     DWORD PTR _temp$[ebp], 0

; 13   :   for(i=0;i<num;i++) temp=temp+d[i];

        mov     DWORD PTR _i$[ebp], 0
        jmp     SHORT $L1298
$L1299:
        mov     eax, DWORD PTR _i$[ebp]
        add     eax, 1
        mov     DWORD PTR _i$[ebp], eax
$L1298:
        mov     ecx, DWORD PTR _i$[ebp]
        cmp     ecx, DWORD PTR _num$[ebp]
        jge     SHORT $L1300
        mov     edx, DWORD PTR _i$[ebp]
        mov     eax, DWORD PTR _d$[ebp]
        mov     ecx, DWORD PTR _temp$[ebp]
        add     ecx, DWORD PTR [eax+edx*4]
        mov     DWORD PTR _temp$[ebp], ecx
        jmp     SHORT $L1299
$L1300:

; 14   :   temp=temp/num;

        mov     eax, DWORD PTR _temp$[ebp]
        cdq
        idiv    DWORD PTR _num$[ebp]
        mov     DWORD PTR _temp$[ebp], eax

; 15   :   return(temp);

        mov     eax, DWORD PTR _temp$[ebp]

; 16   : }

```

```

        pop        edi
        pop        esi
        pop        ebx
        mov        esp, ebp
        pop        ebp
        ret        0
?mean@@YAJQAJJ@Z ENDP        ; mean

```

首先，这部分列表文件为 2 个实参和 2 个局部变量定义了符号常量，用于增加列表文件的可读性。这些变量都保存在堆栈帧中，通过 EBP+8 访问第一个参数（依次再加 4，访问后续参数），通过 EBP-4 访问第一个局部变量（依次再减 4 访问后续局部变量）。

接着，对应源程序的第 11 行、函数开始的花括号（列表文件的表示是“; 11 : {”}）创建函数的起始代码，包括寄存器保护（没有保护 ECX 和 EDX），预留局部变量的堆栈区域。预留 72 字节空间（对应 18 个 4 字节长整型变量），足够 2 个局部变量使用，还有余量。预留的堆栈空间使用串存储指令“REP STOSD”全部填入 CCH。CCH 是断点中断调用指令（INT 3）的机器代码。设置多余的堆栈空间并填入断点中断调用指令是用于防止堆栈错误。因为如果出现非法操作，程序进入堆栈预留空间，执行了断点中断调用指令就将终止程序执行，不至于导致执行非法指令破坏系统。

源程序第 12 行，定义局部变量并设置 TEMP 初值为 0，汇编程序使用一条 MOV 指令将 0 传送到事先预留的堆栈空间中。

源程序第 13 行是循环语句实现数组元素求和。这条语句生成的汇编语言代码中，先设置计数变量 i 初值为 0，然后转移到标号 \$L1298 处；判断当前 i 没有超过元素个数，则实现数组元素相加。这时，用 EDX 等于 i 值，EAX 指向数组地址，ECX 等于累加和 temp 值。完成一个元素求和，转移到标号 \$L1299，实现计数变量 i 的增量，并重复求和过程，直到将所有数组元素都进行了求和。

第 14 行用除法指令 IDIV 求平均值，EAX 得到除法的商（即是平均值），先暂存临时变量 temp，然后又取出送 EAX 作为函数的返回值，以对应 C++ 程序第 15 行的 return 语句。

最后，函数结束的花括号对应源程序第 16 行，生成子程序的结尾代码，恢复寄存器，返回调用程序，其中关键是恢复 ESP 值，保证指向正确的返回地址。

对照汇编语言代码和图 8-1，我们可以清晰地看到函数执行过程中创建的堆栈帧，由此体会其重要作用。例如，如果错误地设置了局部变量，使其覆盖了返回地址、保护的寄存器等，就会导致堆栈溢出，程序不能正确返回到调用程序。

C 语言的许多常用库函数（如 gets、strcpy 等）没有对数组越界加以判断和限制，利用超长的字符数组就可能导致建立在堆栈中的缓冲区溢出，即覆盖缓冲区之外的区域，这就是所谓的“缓冲区溢出”漏洞。如果利用这个漏洞，精心设计一段入侵程序代码，则是臭名昭著的缓冲区溢出攻击。

3. 高级语言的发布版本

从我们掌握的汇编语言知识来看，调试（Debug）版本的程序使用主存作为局部变量，伴随着大量主存读写操作，比之使用寄存器显然性能略差。这是因为调试版本的可执行文件是没有经过优化的。

Visual C++ 使用的编译程序 CL.EXE 支持许多优化参数，如以 O 开头的参数都是优化参

数。在项目配置采用调试版本时，默认不进行优化，对应参数“/Od”。在项目配置采用发布（Release）版本时，对应参数“/O2”，按照最快运行速度的原则进行优化（Maximize Speed）。参数“/O1”是按照最小空间的原则优化（Minimize Size）。它们都可以通过 Visual C++集成开发环境的工程（Project）菜单的设置（Setting）命令进行设置。另外，编译程序还支持针对处理器特性的优化。例如，参数“/G3”是为 80386 处理器进行优化，参数“/G4”是为 80486 处理器进行优化，参数“/G5”是为 Pentium 处理器进行优化，参数“/G6”是为 Pentium Pro 处理器进行优化，包括 Pentium II、Pentium III 和 Pentium 4。Visual C++ .NET 还新增参数“/G7”表示为 Pentium 4 或 AMD Athlon 处理器进行优化，“/GL”参数表示进行整个程序的优化。

现在执行创建菜单的设置活动配置（Set Active Configuration）命令选择发布（Release）版本、重新进行编译和连接，生成经过编译器优化的发布版本的可执行文件。

同样，通过列表文件获得汇编语言代码，我们关注的求平均值函数的部分如下：

```

_d$ = 8
_num$ = 12
?mean@@@YAJQAJJ@Z PROC NEAR    ; mean, COMDAT

; 11    : {

                push     esi

; 12    :   long i,temp=0;
; 13    :   for(i=0;i<num;i++) temp=temp+d[i];

                mov      esi, DWORD PTR _num$[esp]
                xor      eax, eax
                test     esi, esi
                jle      SHORT $L1300
                mov      ecx, DWORD PTR _d$[esp]
                push     edi
                mov      edx, esi

$L1298:
                mov      edi, DWORD PTR [ecx]
                add      ecx, 4
                add      eax, edi
                dec      edx
                jne      SHORT $L1298
                pop      edi

$L1300:

; 14    :   temp=temp/num;

                cdq
                idiv     esi
                pop      esi

; 15    :   return(temp);

```

```

; 16    :}

ret      0
?mean@@YAJQAJJ@Z ENDP      ; mean

```

首先看到：堆栈帧没有按照常规使用 EBP 访问，而是直接利用 ESP，节省 EBP 操作指令是为了提高性能。程序中，ESI 保存数组元素个数，ECX 保存数组地址。

其次看到：局部变量似乎不见了，但实际上是直接使用寄存器实现了它们的功能。EAX 寄存器保存累加和，起到了 temp 局部变量的作用。EDX 寄存器先被赋值数组元素的个数，循环减量，起到了计数变量 i 的作用。

接着分析程序代码：先对 ESI 进行测试，用 JLE（即 JNG）指令排除了元素个数为 0 的特殊情况。此处的 JLE 指令与 JE 指令的功能相同。标号 \$L1298 后面的 5 条指令是循环体，比起调试版本的循环体部分要简单得多，性能自然也提高了。通过对比可发现直接使用汇编语言编写的循环体（例 8-1 和例 8-2），显然优于调试版本，不比发布版本差（至少阅读性更好些）。

最后看到：由于除法指令的结果是平均值，已经在 EAX 中，符合返回值的使用约定，所以不再需要指令实现 return 语句。

4. 调用规范

过程调用中，程序设计语言的调用规范主要约定了过程名（用于连接程序识别）、参数压入堆栈的顺序和平衡堆栈的程序。

Visual C++ 语言的 3 种调用规范，参见表 8-2。可以观察采用 “_cdecl” 调用规范的例题 8-4 的反汇编代码。对应主程序调用语句 “mean(long d[], long num)”，将右边（后边）最后一个参数 num 第 1 个压入堆栈，左边（前边）第 1 个参数最后压入堆栈。主程序增量 ESP 值，平衡堆栈。但函数名不是变为 “_mean”，而是增加了很多奇怪的字符，这是因为 Visual C++ 要对函数名进行修饰，以包含函数名、函数的参数类型、函数的返回类型等诸多信息。可以通过在声明语句中加上 “extern "C"” 去掉 Visual C++ 对函数名的修饰，也就是采用 C 语言修饰格式，此时函数名就是 “_mean”。所以，在前面介绍模块连接进行混合编程的注意事项中，特别要求使用 “extern "C"” 声明 C++ 函数。

表 8-2 Visual C++ 的调用规范

调用规范	_cdecl	_fastcall	_stdcall
命名约定	名字前加下画线	名字前、后都加一个@，后再跟表示参数所占字节数的十进制数值	名字前加下画线，名字后跟@和表示参数所占字节数的十进制数值
参数传递顺序	从右到左	利用 ECX、EDX 传递前 2 个双字参数，其他参数再通过堆栈传递（从右到左）	从右到左
平衡堆栈的程序	调用程序	被调用程序	被调用程序

MASM 汇编语言支持 6 种语言类型，见表 8-3。其中，C、SYSCALL 和 STDCALL 语言类型支持长度可变的参数 (VARARG)，PASCAL、BASIC 和 FORTRAN 语言类型还保存 EBP。

同样，可以将采用 C 语言类型的例 7-5 列表文件的汇编代码与表 8-3 的 C 语言类型规范进行对照。另外，可以观察调用 API 函数时所采用的名称，如第 7 章 Windows 应用程序退出语句 “invoke ExitProcess,0” 的反汇编代码如下：

```
push 0
call _ExitProcess@4
```

可表明它采用的是 STDCALL 语言类型。

表 8-3 MASM 6.x 的语言类型

语言类型	C	SYSCALL	STDCALL	PASCAL	BASIC	FORTRAN
命名约定	名字前加下画线		名字前加下画线	名字变大写	名字变大写	名字变大写
参数传递顺序	从右到左	从右到左	从右到左	从左到右	从左到右	从左到右
平衡堆栈的程序	调用程序	被调用程序	被调用程序 ^(注)	被调用程序	被调用程序	被调用程序

注：STDCALL 如果采用 VARARG（长度可变）参数类型，则是调用程序平衡堆栈，否则是被调用程序平衡堆栈。

5. ENTER 和 LEAVE 指令

ENTER 和 LEAVE 是 IA-32 处理器为支持堆栈帧而设计的指令。

(1) ENTER 指令建立堆栈帧

ENTER i16,i8 ;i16 是在堆栈分配的字节数，i8 为过程的嵌套层次

在嵌套层次为 0 时，“ENTER i16, 0”指令的作用是：

- ⊙ EBP 压入堆栈，对应指令 PUSH EBP。
 - ⊙ 设置 EBP 等于 ESP，对应指令 MOV EBP, ESP。
 - ⊙ ESP 减去 i16，对应指令 SUB ESP, i16（或者 ADD ESP, -i16）。
- 当嵌套层次大于等于 1 时，过程有嵌套，需要重复进栈和调整 EBP 数值。

(2) LEAVE 指令释放堆栈帧

LEAVE ;释放对应 ENTER 指令建立的堆栈帧

它的功能是：

- ⊙ 设置 ESP 等于 EBP，对应指令 MOV ESP, EBP。
- ⊙ EBP 弹出堆栈，对应指令 POP EBP。

所以，使用 ENTER 指令进入过程建立堆栈帧，过程结尾处使用 LEAVE 指令释放对应的堆栈帧，可以省去多条指令，简化编程。不过，ENTER 指令是一条复杂的指令，在 Pentium 及其后的处理器中开始使用先进技术加快多条简单指令的执行，ENTER 指令也用得越来越少。

如果过程使用 ENTER 指令，建议结尾一定使用 LEAVE 指令，这样配合起来不会出错。由上可见，堆栈对于程序来说非常重要，所以程序要为堆栈留出足够的空间。

8.3 调用高级语言函数

混合编程中，高级语言调用汇编语言的子程序比较常见。当然，汇编语言也可以调用高级语言函数，包括调用 Windows API 函数、C 标准库函数等。调用的注意事项与 8.2 节所述的约定一致。例如，汇编程序调用的 C++ 函数使用 “extern “C”{ }” 进行定义。

如果是单独汇编的汇编语言程序或者汇编语言模块调用高级语言函数（8.3.2 节示例），就可以使用 PROTO 声明函数（C++ 函数选择 C 语言类型、Windows API 函数选择 STDCALL 语言类型），并使用 INVOKE 调用函数。

如果是在嵌入汇编语句中调用高级语言函数（8.3.1 节示例），则不能使用 PROTO 声明，也不能使用 INVOKE 调用，因为 Visual C++ 6.0 并不支持这些伪指令。这时只能使用 CALL 指令调用，并按照调用规范压入堆栈参数、平衡堆栈。

8.3.1 嵌入汇编程序中调用高级语言函数

下面例题程序显示一个消息窗口，实现与例 7-3 一样的功能。

【例 8-5】 嵌入汇编中调用消息窗口函数程序。

```
#include "windows.h"
int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine, int iCmdShow)
{
    char *lpCaption ="欢迎";
    char *lpText ="你好，汇编语言! ";
    __asm{
        push MB_OK           //MessageBox 参数入栈
        push lpCaption
        push lpText
        push NULL
        call dword ptr MessageBox //调用 MessageBox，API 函数不需要调整 ESP
    }
    return 0;
}
```

这是一个 Windows 图形界面程序，开发时请选择 32 位窗口应用程序(Win32 Application) 类型。使用 CALL 指令调用 API 函数前，按照 STDCALL 语言类型从右到左依次压入参数，调用后不必增量 ESP。

8.3.2 汇编程序中调用 C 库函数

C 语言有一系列的标准函数，称为标准 C 库 (Standard C Library)。标准 C 库的函数在 C++ 程序中同样可用，汇编程序也可以调用。例如，printf 函数和 scanf 函数是 C 语言常用的格式化输出和输入函数，它们存放在 MSVCRT.DLL 动态链接库中，开发时需要使用导入库 MSVCRT.LIB。

1. printf 函数

printf 函数实现格式化输出，C 语言原型是：

```
int printf(const char *,...);
```

该函数的第一个参数是字符串指针，后面的参数是输出值，个数不定。

对应的汇编语言声明可以是：

```
printf      proto c,:ptr byte,:vararg
```

其中，“PTR BYTE”说明是字节变量的指针，也可以用 DWORD 类型。

【例 8-6】 调用 C 库函数输出信息程序。

```
.686
.model flat,stdcall
option casemap:none
includelib bin32\kernel32.lib
ExitProcess proto,:dword
includelib bin32\msvcrt.lib
```



```

printf    proto c, :ptr byte, :vararg
          .data
msg       db 'Hello, World!', 0dh, 0ah, 0
          .code
start:
          invoke printf, addr msg
          invoke ExitProcess, 0
          end start

```

本书配套软件在 BIN32 子目录下保存 MSVCRT.LIB 文件。

本例程序使用汇编语言实现了经典的“printf "Hello, World!\n"”语句功能。

2. scanf 函数

scanf 函数实现格式化输入，C 语言原型是：

```
int scanf(char *,...);
```

该函数的第一个参数是字符串指针，后面的参数保存输入值的变量地址，个数不定。

对应的汇编语言声明是：

```
scanf    proto c, :ptr byte, :vararg
```

【例 8-7】 格式化输入输出程序。

利用 scanf 输入一个实数，然后用 printf 以十六进制整数形式输出，观察浮点格式的编码（参见 9.1 节）。

```

.686
.model flat, stdcall
option casemap:none
includelib bin32\kernel32.lib
ExitProcess proto, :dword
includelib bin32\msvcrt.lib
printf      proto c, :ptr byte, :vararg
scanf       proto c, :ptr byte, :vararg
          .data
msg1        db 'Enter a real number: ', 0
format1     db '%f', 0
var         dd ?
msg2        db 'The codes in machine:  %X', 0dh, 0ah, 0
          .code
start:
          invoke printf, addr msg1
          invoke scanf, addr format1, addr var
          invoke printf, addr msg2, var
          invoke ExitProcess, 0
          end start

```

8.4 使用 Visual C++ 开发环境

功能强大的集成开发环境 Visual C++ 不仅支持 C、C++ 语言程序的开发，还能够用来编辑、汇编、连接和调试汇编语言程序。下面简单描述开发和调试过程，并说明所应该注意的

问题。

事实上，微软已经不再单独升级 MASM，而是配套 Visual C++ 使用。具有了高级语言特性的 MASM 也不再是一个简单的汇编程序，而是成为了 C、C++ 语言的辅助工具。

8.4.1 汇编语言程序的开发过程

可以选择第 7 章例题程序（如例 7-1～例 7-4）进行实践，下面说明其操作步骤。如果是开发例 8-1、例 8-2、例 8-4 和例 8-5 程序，则不需要设置汇编命令这个步骤。

（1）创建工程项目

执行文件（File）菜单的新建（New）命令，新建一个工程项目（Project）。根据需要选择 32 位控制台应用程序（Win32 Console Application）或 32 位窗口应用程序（Win32 Application）。输入工程项目所在的磁盘目录，输入工程名称（如例 7-1 取名 eg701），如图 8-2 所示，确认后选择创建一个空白工程（An Empty project）。



图 8-2 创建工程项目

（2）创建源程序文件并加入工程项目

执行“文件”菜单的“新建”命令，新建一个源程序文件。选择文本文件（Text file），输入源程序文件名以及扩展名（如 eg701.asm），汇编源程序文件使用扩展名.ASM，C++源程序文件使用扩展名.CPP。默认情况下，“添加到工程”是被选中状态，这个文件被加入工程项目。

如果已有源程序文件，则可以将该文件复制到该工程项目所在的磁盘目录下，但需要加入到该工程项目。这可以通过工程（Project）菜单、执行添加到工程（Add To Project）命令、展开文件（Files）对话框进行添加。

也可以通过“工程”菜单的“添加到工程”命令，展开新建对话框，在该工程项目中进行源文件的创建。

用汇编语言编写 32 位控制台或窗口应用程序，采用了 Windows 的 API 函数。由于 Visual C++ 环境已经具有导入库文件，所以汇编语言程序中子程序库包含语句 INCLUDELIB 就不需要了（Visual C++ 的导入库文件在其 LIB 目录下）。如果源程序利用 INCLUDE 语句指明 Windows 常量和 API 函数声明所在的包含文件，那么在 Visual C++ 环境中也要确定其路径。可以在源程序中给出绝对路径，也可以将这些包含文件复制到 Visual C++ 头文件所在的目录 INCLUDE 的某个子目录下，源程序中使用相对路径指明该子目录（和文件）即可。

（3）设置汇编命令

在 Visual C++ 集成环境左边选择文件视图（FileView），并选中汇编源程序文件；然后选择快捷菜单的设置（Settings）命令，或者通过“工程”菜单的“设置”命令展开其工程设置

窗口。

在工程设置窗口的右边选择定制创建（自定义组建，Custom Build）标签，在其命令（Commands）文本框中输入进行汇编的命令，如例 7-1 的汇编命令 “ml /c /coff eg701.asm”，还可以带上参数 “/Fl” 生成列表文件、参数 “/Zi” 加入调试信息。

在工程设置窗口的“自定义组建”标签中，在其输出文本框中输入汇编后的目标模块文件名，对应例 7-1 输入 “eg701.obj”，如图 8-3 所示。

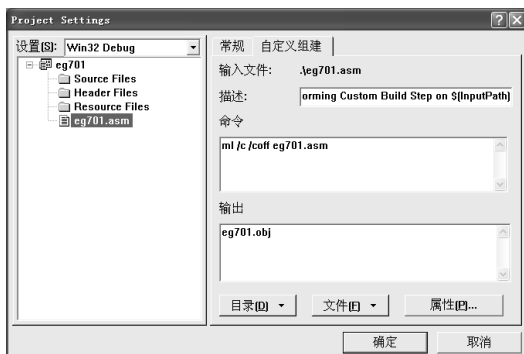


图 8-3 设置汇编命令

另外，应该事先将 ML.EXE 和 ML.ERR 文件复制到 Visual C++ 所在的 Bin 目录下，或者在输入汇编命令时同时输入 ML.EXE 所在的目录路径。

（4）进行汇编、连接生成可执行文件

这时可以调用“创建”菜单的“创建”命令进行汇编语言程序的汇编和连接。汇编连接的有关信息显示在下面输出窗口的创建视图中。如果程序正确无误，会生成可执行文件（默认是调试版本，在 DEBUG 目录下）。如果源程序有错误，创建视图将显示错误所在的行号以及错误的原因。双击该错误信息，光标将定位到出现错误的源程序行。

使用 Visual C++6.0 版本开发汇编语言程序的关键步骤是设置汇编命令。因为每个源程序文件名不同，所以每个程序的开发过程中都需要进行设置，比较烦琐。一个简单的处理方法是汇编语言源程序文件都使用同一个名字，开发完成后再进行重命名。这样，只要进行一次设置就可以了。以后开发的过程也变得非常简单，只要将汇编源程序文件改名，复制到原来创建的目录中，然后使用“创建”命令就可以进行汇编、连接生成可执行文件。

8.4.2 汇编程序的调试过程

Visual C++ 集成开发环境包含 Windows 应用程序的调试程序，不仅可以调试高级语言的程序，也可以调试汇编程序。调试高级语言源程序时，还可以对其进行反汇编，实现汇编语言级的调试。下面结合例 8-4 简单说明其过程，不论调试 C++ 程序还是汇编程序，过程类似。当然，要进行源程序级的调试，需要带入调试信息，高级语言则是调试（Debug）版本。

（1）设置汇编语言的调试选项

为了使得 Visual C++ 集成开发环境更适合对汇编语言程序的调试，可以通过“工具”（Tools）菜单的“选项”（Options）命令展开调试（Debug）标签页进行设置。常规（General）框下的“十六进制”显示（Hexadecimal Display）应该选中，以便以十六进制形式显示输入/输出数据（此时可以用 0n 开头表示输入十进制数据）。反汇编窗口（Disassembly window）

框下要选中“代码字节”(Code bytes)。存储器窗口(Memory window)框下选中“定宽”(Fixed width),并在后面填入数字 16,如图 8-4 所示。



图 8-4 设置汇编语言的调试选项

(2) 设置断点，进行断点调试

断点(Breakpoint)是让程序调试过程中暂停执行的语句，以便观察该语句之前的运行状态或当前结果，用于判断在此之前程序是否运行正常。

在“文件视图”(FileView)中双击源程序文件名，则编辑窗口将显示这个源程序。移动光标到需要暂停的语句行，按 F9 键(或者单击工具栏上的手形图标)，这样就在该行设置了一个断点(前面有一个红色的圆点)；光标在已经设置断点的语句行时再次按 F9 键，则取消断点。在反汇编窗口中，可以针对指令进行断点设置。一个程序可以设置多个断点。

使用“创建”菜单的“执行”命令(快捷键是 Ctrl+F5)，可以运行已经编译连接的可执行程序。如果要进行调试，需要从“创建”菜单的“开始调试”命令中选择在调试状态下执行程序，如“运行”(Go，其快捷键是 F5)命令。如果程序设置了断点，启动程序运行后将停留到断点语句行，在源程序窗口有一个黄色箭头指示。

如果不设置断点，也可以将光标移动到要暂停执行的语句前，然后选择“执行到光标”(Run to Cursor)命令进行断点调试。

进入调试状态后，原来的“创建”菜单也改变成为了“调试”菜单。这时，利用“视图”(View)菜单的“调试窗口”(Debug Windows)命令，就可以打开各种窗口观察程序当前的运行状态。

现在调试例 8-4，首先打开该工程项目，完成调试版本的开发，生成可执行文件。接着打开源程序文件，在主函数输出语句前设置断点，按 F5 键开始调试，程序会在该语句前暂停。查看反汇编(Disassembly)窗口，其中就是实际执行的汇编代码，如图 8-5 所示。

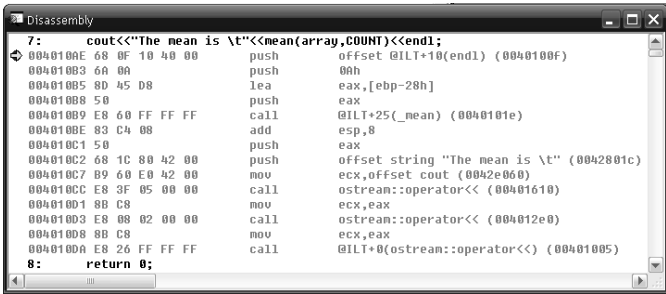


图 8-5 输出语句的反汇编窗口

这时可以查看存储器（Memory）窗口，在地址（Address）栏输入变量名，下面就显示该变量所在的主存地址，十六进制形式的数值和 ASCII 码字符；右键单击之，还可以选择字或双字显示形式。另外，寄存器（Register）窗口显示处理器的寄存器内容，变量（Variable）窗口显示当前函数的变量，监视（Watch）窗口可以输入需要观察的变量或寄存器名，变量或寄存器内容都可以在这些窗口中直接改变。

（3）单步调试

如果需要仔细观察每条语句的执行情况，可以采用单步调试。执行单步调试命令，则程序执行一条语句，就自动暂停（好像每条语句都被设置了断点一样）。对汇编语言来说，一条语句对应一条指令。所以，如果当前激活的窗口是反汇编窗口，则单步执行时每条指令均暂停；而高级语言的源程序文件窗口是当前激活窗口时，则是每条语句暂停。

单步调试命令分成两种：

- ④ 不跟踪子程序的单步执行（Step Over，快捷键是 F10）——只进行主程序（C 语言称主函数）的单步调试。也就是说，当遇到调用子程序（C、C++语言称函数）语句时，完成子程序执行并返回到调用语句的下一条语句暂停，不跟踪子程序的每条语句。
- ④ 跟踪子程序的单步执行（Step Into，快捷键是 F11）——进行子程序语句的单步调试。也就是说，当遇到调用子程序语句时，进入到子程序的第一条语句暂停，可以进入子程序当中进行调试，跟踪子程序的每条语句执行情况。在子程序中可以执行单步跳出（Step Out）命令结束单步调试，完成子程序执行并返回主程序。

接着前面的例 8-4 调试，程序现在暂停在主函数的输出语句前，按 F11 键进行单步调试，程序进入 mean 函数，并暂停在花括号前。查看此时的反汇编窗口，如图 8-6 所示。

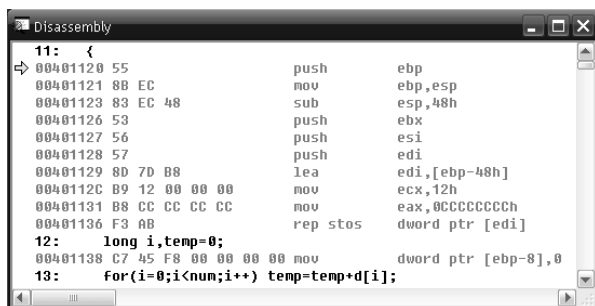


图 8-6 函数建立堆栈帧的反汇编窗口

可以继续断点或者单步调试，观察程序运行情况。例如，让例 8-4 程序在循环语句前暂停，打开存储器窗口，在地址栏输入堆栈指针寄存器名 ESP，函数的堆栈帧就一目了然了，可以右击选择双字显示格式（Long Hex Format），以便观察。

最后，执行停止调试（Stop Debugging）命令，退出调试状态。

（4）汇编语言程序的调试

不论是在 Visual C++环境开发的汇编语言程序，还是利用本书提供的简易 MASM 环境开发的汇编语言程序，都可以使用 Visual C++的调试程序。

打开 Visual C++集成环境，执行“文件”菜单的“打开”命令，选择已经生成的可执行文件，然后按 F11 快捷键，或者展开“创建”菜单的“开始调试”命令，选择跟踪子程序的“单步执行”命令，就进入调试状态，并暂停在程序开始位置。如果调试信息完整，源程序文件会被自动打开，基本调试方法就与高级语言一样了。

如果只有可执行文件，没有生成调试信息或没有调试信息，也可以进行汇编语言级的调试。打开可执行文件，按 F11 键进入调试状态。调试程序会提示该可执行文件没有调试信息，单击“确定”按钮，反汇编窗口自动弹出，程序在起始指令处暂停。接着就可以设置断点或单步调试了，由于没有调试信息，会受到一些限制。

上面只是简单描述了调试的一般过程，要熟练掌握程序调试需要反复实践。对比前 5 章使用的 DEBUG 调试程序，可以发现 Visual C++ 的调试程序虽然功能强大、操作灵活，但更加复杂，掌握起来似乎也更困难。不过，单步调试和断点调试仍然是最基本的方法，查看寄存器、变量和堆栈等都是常用的判断依据。

习 题 8

8.1 简答题

- (1) 什么是混合编程？
- (2) 混合编程有什么优势？
- (3) 汇编语言与 C、C++ 语言的混合编程有哪两种方法？
- (4) 进行模块连接的混合编程，一般要注意哪些方面的约定规则？
- (5) C++ 函数通过什么方式传递入口参数？
- (6) 堆栈帧是一个什么作用的堆栈空间？
- (7) MASM 使用 INVOKE 伪指令方便调用高级语言函数，在嵌入汇编代码中也能够这样使用吗？
- (8) 在 Visual C++ 环境中开发汇编语言程序，为什么可以不用包含导入库文件？
- (9) 什么是断点调试？
- (10) 什么是单步调试？

8.2 判断题

- (1) C++ 程序中嵌入汇编指令，但嵌入汇编中不能使用汇编语言的注释形式。
- (2) 汇编语言的宏很有特色，所以仍然可以应用于嵌入汇编中。
- (3) 嵌入汇编中可以直接使用 C++ 语言定义的变量。
- (4) 嵌入汇编语句仍然可以利用 OFFSET 获得全部变量的地址。
- (5) MASM 汇编语言的 C 语言类型对应 C++ 语言的 `_cdecl`。
- (6) 局部变量是通过堆栈创建的。
- (7) 使用寄存器替代频繁访问的变量，可以提升程序性能。
- (8) Visual C++ 的发布版本相对于调试版本来说，只是去掉了调试信息，其他一样。
- (9) 汇编语言可以调用 C 库函数，且不需要导入库文件。
- (10) 没有调试信息，调试程序无法进行汇编语言级的调试。

8.3 填空题

- (1) 有一个数据 100，要在嵌入汇编指令中作为立即数，且用十六进制形式表达，可以像汇编语言中一样表达为_____，也可以像 C++ 语言一样表达为_____。
- (2) C++ 中有一个整型 (int) 数组 array，要在嵌入汇编语句中操作数组元素 array[4]，可以表达为_____。
- (3) 有一个采用 C 语言类型的汇编语言子程序，如果 C++ 程序中要调用它，声明函数时

要增加_____修饰符。

(4) 函数调用中, 通常通过 EBP 指向堆栈帧, 其值减_____访问第一个局部变量, 其值加_____访问第一个入口参数, 返回地址则由其值加_____指向。

(5) C++函数返回一个 32 位整数, 返回值使用_____保存。

(6) C++语言的 char、short 和 long 变量类型对应汇编语言的类型依次是_____, _____和_____。

(7) 某个采用 “_cdecl” 调用规范的 C++函数 sum(int array[], int num), 先压入堆栈的参数是_____。平衡堆栈需要将 ESP 加_____, 由_____程序实现。

(8) 反汇编代码对调用程序退出 API 函数使用 “_ExitProcess@4” 名称, 其中的 “4” 表示_____。

(9) LEAVE 指令用于子程序返回前, 相当于_____和_____指令的功能。

(10) printf 函数支持个数不定的参数, 在使用 PROTO 进行声明时需要采用类型符号_____。

8.4 阅读如下嵌入汇编的 C++程序, 说明显示结果。

```
#include <iostream.h>
int power2(int,int);
void main(void)
{
    cout<<power2(5,6)<<endl;
}
int power2(int num,int power)
{
    __asm
    {
        mov eax,num
        mov ecx,power
        shl eax,cl
    }
}
```

8.5 阅读如下程序, 说明输出结果。

```
// C++程序
#include <iostream.h>
extern "C" { void MLSub(char *,short *,long *);}
char chararray[4] = "abc";
short shortarray[3] = {1,2,3};
long longarray[3] = {32768,32769,32770};
void main(void)
{
    cout<<chararray<<endl;
    cout<<shortarray[0]<<shortarray[1]<<shortarray[2]<<endl;
    cout<<longarray[0]<<longarray[1]<<longarray[2]<<endl;
    MLSub (chararray,shortarray,longarray);
    cout<<chararray<<endl;
    cout<<shortarray[0]<<shortarray[1]<<shortarray[2]<<endl;
```

```

        cout<<longarray[0]<<longarray[1]<<longarray[2]<<endl;
    }
};汇编语言程序
        .686
        .model flat,c
MLSub    proto ,:dword,:dword,:dword
        .code
MLSub    proc uses esi,arraychar:dword,arrayshort:dword,arraylong:dword
        mov esi,arraychar
        mov byte ptr [esi],"x"
        mov byte ptr [esi+1],"y"
        mov byte ptr [esi+2],"z"
        mov esi,arrayshort;
        add word ptr [esi],7
        add word ptr [esi+2],7
        add word ptr [esi+4],7
        mov esi,arraylong
        inc dword ptr [esi]
        inc dword ptr [esi+4]
        inc dword ptr [esi+8]
        ret
MLSub    endp
        end

```

8.6 如下 C++程序中输入了两个整数，然后调用汇编语言子程序对这两个数求积，在主程序中打印计算结果。编写汇编语言子程序模块。

```

#include <iostream.h>
extern "C" { int multi(int x,int y);}
void main(void)
{
    int x,y;
    cin>>x;
    cin>>y;
    cout<< multi(x,y)<<endl;
}

```

8.7 堆栈帧的创建步骤一般有哪些？如果进行代码优化，还一定遵循这个原则吗？

8.8 求最大公约数程序。

最大公约数（Greatest Common Divisor）是能够同时被两个（多个）无符号数整除的最大整数。求最大公约数（假设为 M 和 N ， $M > N$ ）通常使用辗转相除法，其过程是：

- (1) 用 M 除以 N ，得到余数 R 。
- (2) 若余数不等于 0，则 $M \leftarrow N$ ， $N \leftarrow R$ ，继续上述除法求余数。
- (3) 若余数等于 0，则 N 就是公约数。

C、C++程序可以编写如下：

```

while(r!=0)
{
    r = m % n;
}

```



```

        m = n;
        n = r;
    }

```

采用扩展过程定义 PROC 伪指令编写求最大公约数子程序，入口参数是两个 32 位无符号整数，返回值是最大公约数。

使用 Visual C++编写主程序，从键盘输入两个无符号整数，调用该子程序，最后输出最大公约数。

8.9 如下 C++程序实现对小于指定数值 (sample) 的数组 (array) 元素累加求和的功能。

```

#include <iostream.h>
long sums(long array[],long count,long sample);
void main()
{
    long array[]={10,40,80,50,99,76,15,57,88,20};
    long sample=60;
    long count=sizeof array /sizeof sample;
    cout<<sums(array,count,sample)<<endl;
}
long sums(long array[],long count,long sample)
{
    long i=0,sum=0;
    while(i < count)
    {
        if (array[i] <= sample)
        {
            sum += array[i];
        }
        i++;
    }
    return(sum);
}

```

(1) 在 Visual C++开发环境生成调试版本的含汇编语言的列表文件、可执行文件，并据此画出函数 sums 的堆栈帧。

(2) 在 Visual C++开发环境生成发布版本的含汇编语言的列表文件、可执行文件，并说明编译程序的优化方法。

(3) 使用汇编语言，配合 MASM 的高级特性伪指令 (扩展定义 PROC、IF 和 WHILE) 编写实现该程序功能，并生成完整的列表文件，与高级语言的发布版本比较编程优劣。

(4) 使用汇编语言，不采用 MASM 的高级语言特性编写实现该程序功能，尽量编写一个最优化的程序。

8.10 按照例 5-12 功能的要求，编写汇编语言程序，调用 printf 函数实现将二进制代码 01100100 以二进制、十六进制、十进制和字符形式输出。提醒，printf 函数不支持二进制数输出，需要处理后再输出。

第9章 浮点、多媒体及64位指令

指令系统（指令集）是处理器支持的所有指令的集合。作为复杂指令集计算机 CISC 的典型代表，为了保证软件向后兼容（现在的软件能在后续产品上继续运行），IA-32 处理器维持了一个庞大的指令系统，如表 9-1 所示。

表 9-1 IA-32 处理器指令系统

指令类型	指令特点
通用指令	处理器的基本指令，包括整数的传送和运算、流程控制、输入/输出、位操作等
浮点指令	浮点数处理指令，包括浮点数的传送、算术运算、超越函数运算、比较、控制等
多媒体指令	多媒体数据处理指令，包括 MMX、SSE、SSE2 以及 SSE3 和 SSSE3、SSE4 等
系统指令	为核心程序和操作系统提供的处理器功能控制指令

通用指令属于处理器的基本指令，主要处理整数、地址和 BCD 码数据类型，包括数据传送、算术逻辑运算、程序流程控制、外设输入/输出等指令类型，是编写应用程序和系统程序主要的、必不可少的指令。本书前面各章从不同角度学习了 IA-32 处理器通用指令当中的最基本、最常用的指令。

IA-32 处理器的通用指令绝大多数都是在 16 位 8086 处理器基本指令（包括 80186 完善的若干指令）的基础上扩展形成的 32 位指令。80286 开始陆续增加的通用指令，往往是针对特定应用目的而设计的，多数是不常用的复杂指令。

80286 引入了保护方式，所以主要增加了用于保护方式的指令。这些指令多数属于所谓的特权指令，通常只有系统核心程序能够使用它们，是主要的系统指令。

80386 在执行单元新增了一个“桶型”移位器（实现快速移位操作的硬件电路），所以新增许多与位操作有关的指令。80486、Pentium 和 Pentium Pro 都增加了若干整数指令，以增强某个方面的处理功能。80486 开始的 IA-32 处理器芯片上集成有浮点处理单元 FPU，所以都支持浮点数据的处理指令。

Pentium II~Pentium 4 处理器陆续增加了处理整型多媒体数据的 MMX 指令、单精度浮点型多媒体数据的 SSE 指令、双精度浮点型多媒体数据的 SSE2 指令以及完善多媒体数据处理的 SSE3 指令等。支持 Intel 64 位结构的 Pentium 4 处理器，还提供了 64 位指令，并具有虚拟机管理指令。

本章简要介绍基本的浮点指令、多媒体指令和 64 位指令的特点，目的是为了使大家对这些指令有所了解。

9.1 浮点指令

简单的数据处理、实时控制领域一般使用整数，所以传统的处理器或简单的微控制器只有整数处理单元。实际应用当中还要使用实数，尤其是科学计算等工程领域。有些实数经过移动小数点位置，可以用整数编码表达和处理，但可能损失精度。实数也可以经过一定格式转换后，完全用整数指令仿真，但处理速度难尽如人意。在计算机中表达实数采用浮点数据

格式，配合浮点指令进行编程。Intel 80x87 是与 Intel 80x86 处理器配合使用的浮点处理器，Intel 80486 及以后的 IA-32 处理器已经集成进了浮点处理单元（Floating-Point Unit），它们统称为 x87 FPU。

9.1.1 实数编码

实数（Real Number）常采用科学表示法表达，如数值“-123.456”可表示为 -1.23456×10^2 。该表示法包括三部分：指数、有效数字两个域和一个符号位。指数用来描述数据的幂，反映数据的大小或量级；有效数字反映数据的精度。在计算机中，表达实数的浮点格式也可以采用科学表达法，只是指数和有效数字要用二进制数表示，指数是 2 的幂（而不是 10 的幂），正负符号也只能用 0 和 1 区别。

另外，实数是一个连续系统，理论上说任意大小与精度的数据都可以表示。但是在计算机中，由于处理器的字长和寄存器位数有限，实际上所表达的数值是离散的，其精度和大小都是有限的。显而易见，有效数字位数越多，能表达数值的精度也就越高；指数位数越多，能表达数值的范围就越大。所以，浮点格式表达的数值只是实数系统的一个子集。

1. 浮点数据格式

计算机中的浮点数据格式如图 9-1 所示，分成指数、有效数字和符号位三部分。IEEE 754 标准（1985 年）制定有 32 位（4 字节）编码的单精度浮点数和 64 位（8 字节）编码的双精度浮点数据格式。

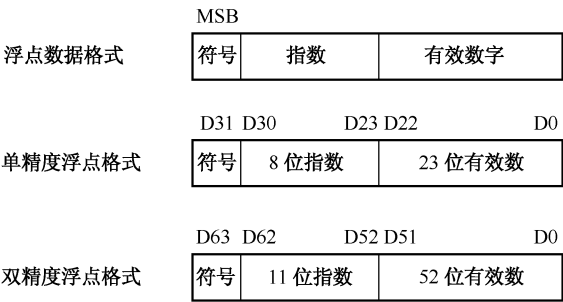


图 9-1 浮点数据格式

符号（Sign）：表示数据的正负，在最高有效位（MSB）。负数的符号位为 1，正数的符号为 0。

指数（Exponent）：也被称为阶码，表示数据以 2 为底的幂，恒为整数，使用偏移码（Biased Exponent）表达。单精度浮点数用 8 位，双精度浮点数用 11 位表达指数。

有效数（Significand）：表示数据的有效数字，反映数据的精度。单精度浮点数用最低 23 位表达有效数，双精度浮点数用最低 52 位表达有效数。有效数一般采用规格化（Normalized）形式，是一个纯小数，所以也被称为尾数（Mantissa）、小数或分数（Fraction）。

2. 浮点阶码

类似补码、反码等编码，偏移编码（简称移码）也是表达有符号整数的一种编码。标准偏移码选择从全 0 到全 1 编码中间的编码作为 0，即从无符号整数的全 0 编码开始向上偏移一半后得到的编码作为偏移码的 0（对 8 位就是 $128 = 10000000B$ ）。以这个 0 编码为基准，

向上的编码为正数，向下的编码为负数。于是， N 位偏移码=真值+ 2^{N-1} 。

例如，对 8 位编码，真值 0 的无符号整数编码是全 0，标准偏移码则表示为 $0+128=00000000\text{B}+10000000\text{B}=10000000\text{B}$ ，恰好是中间的编码。真值 127 的无符号整数编码是 01111111B ，标准偏移码则表示 $127+128=01111111\text{B}+10000000\text{B}=11111111\text{B}$ 。

反过来，采用标准偏移码的真值=偏移码- 2^{N-1} 。例如，对于偏移码全 0 的编码，其真值= $00000000\text{B}-10000000\text{B}=0-128=-128$ 。对比补码，偏移码仅与之符号位相反，参见表 9-2 所示。

表 9-2 8 位二进制数的补码、标准偏移码、浮点阶码

十进制真值	补 码	标准偏移码	浮 点 阶 码
+127	01111111	11111111	11111110
+126	01111110	11111110	11111101
+2	00000010	10000010	10000001
+1	00000001	10000001	10000000
0	00000000	10000000	01111111
-1	11111111	01111111	01111110
-2	11111110	01111110	01111101
-126	10000010	00000010	00000001
-127	10000001	00000001	
-128	10000000	00000000	

为了便于进行浮点数据运算，指数采用偏移编码。但是，在 IEEE 574 标准中，全 0、全 1 两个编码用做特殊目的，其余编码表示阶码数值。所以，单精度浮点数据格式中的 8 位指数的偏移基数为 127，用二进制编码 $0000001\sim 11111110$ 表达 $-126\sim +127$ 。双精度浮点数的偏移基数为 1023。相互转换的公式如下：

单精度浮点数据：真值=浮点阶码-127，浮点阶码=真值+127。

双精度浮点数据：真值=浮点阶码-1023，浮点阶码=真值+1023。

3. 规格化浮点数

十进制科学表示法的实数可以有多个形式，如 $-1.23456\times 10^2=-0.123456\times 10^3=-12.3456\times 10^1$ 。此时，只要小数点左移或右移，相应进行指数增量或减量。在浮点格式中数据也会出现同样的情况。为了避免多样性，同时也为了能够表达更多的有效位数，浮点数据格式的有效数字一般采用规格化形式，它表达的数值是：

1.XXX...XX

由于去除了前导 0，它的最高位恒为 1，随后都是小数部分，这样有效数字只需要表达小数部分，其小数点在最左端，它隐含一个整数 1。这就是通常使用的浮点数据。

【例 9-1】 把浮点格式数据转换为实数表达。

某个单精度浮点数如下：

BE580000H=1011 1110 0101 1000 0000 0000 0000 0000 B

将它分成 1 位符号、8 位阶码和 23 位有效数字三部分：

BE580000H=1 01111100 10110000000000000000000 B

符号位为 1，表示负数。

指数编码是 01111100，表示指数为 $124-127=-3$ 。

有效数字部分是 101100000000000000，表示有效数为 $1.1011\text{ B}=1.6875$ 。

所以，这个实数为 $-1.6875 \times 2^{-3} = -1.6875 \times 0.125 = -0.2109375$ 。

【例 9-2】 把实数转换成浮点数据格式。

对实数“100.25”进行如下转换：

$$100.25 = 0110\ 0100.01\text{B} = 1.10010001\text{B} \times 2^6$$

于是，符号位=0。

指数部分是 6，8 位阶码为 10000101（即 $6+127=133$ ）。

有效数字部分是 100100010000000000000000。

这样，100.25 表示成单精度浮点数为（参见例 9-4 程序的验证）：

$$\begin{aligned} & 0\ 10000101\ 100100010000000000000000\text{B} \\ & = 0100\ 0010\ 1100\ 1000\ 1000\ 0000\ 0000\ 0000\ \text{B} \\ & = 42\text{C}88000\text{H} \end{aligned}$$

4. 非规格化浮点数和机器零

浮点格式的规格化数所表达的实数是有限的。例如，对单精度浮点规格化浮点数，其最接近 0 的情况是：指数最小（-126）、有效数字最小（1.0），即数值 $\pm 2^{-126}$ （ $\approx \pm 1.18 \times 10^{-38}$ ）。当数据比这个最小数还要小，还要接近 0 时，规格化浮点格式无法表示，这就是下溢（Underflow）。

为了能够表达更小的实数，制定了“非规格化浮点数”：它用指数编码为全 0 表示 -126；有效数字仅表示小数部分，但不能是全 0，表示的数值是：

$$0.\text{XXX}\cdots\text{XX}$$

这时，有效数字最小编码是仅有最低位为 1、其他为 0，表示的数值是： 2^{-23} 。这样非规格化浮点数能够表示到 $\pm 2^{-126} \times 2^{-23}$ （ $\approx \pm 1.40 \times 10^{-45}$ ）。

非规格化浮点数表示了下溢，程序员可以在下溢异常处理程序中利用它。

如果数据比非规格化浮点数所能表达的（绝对值）最小数还要接近 0，只能使用机器零表示。机器零的指数和有效数字的编码都是全 0，符号位可以是 0 或 1，所以分成 +0 和 -0。机器零用浮点数据格式表达了真值 0，以及小于规格化数（或非规格化数）（绝对值）最小值的、无法表达的实数。

5. 无穷大

对单精度浮点规格化浮点数，其最大数的情况是：指数最大（127）、有效数字最大（编码为全 1，表达数值为 $1+1-2^{-23}$ ），即数值 $\pm(2-2^{-23}) \times 2^{127}$ （ $\approx \pm 3.40 \times 10^{38}$ ）。当数据比这个最大数还要大时，规格化浮点格式无法表示，这就是上溢（Overflow）。

大于规格化浮点数所能表达的（绝对值）最大数的真值，浮点格式用无穷大表达。它根据符号位分为正无穷大（ $+\infty$ ）和负无穷大（ $-\infty$ ），指数编码为全 1，有效数编码为全 0。

浮点格式通过组合指数和有效数的不同编码，可以表达规格化有限数（Normalized Finite）、非规格化有限数（Denormal Finite）、有符号零（Signed Zero）、有符号无穷大（Signed Infinity），如图 9-2 所示。除此之外，还支持一类特殊的编码：指数编码是全 1、有效数字编

码不是全 0，被称为非数 NaN（Not a Number），因为 NaN 不是实数的一部分。程序员可以利用 NaN 等进行特殊情况的处理。

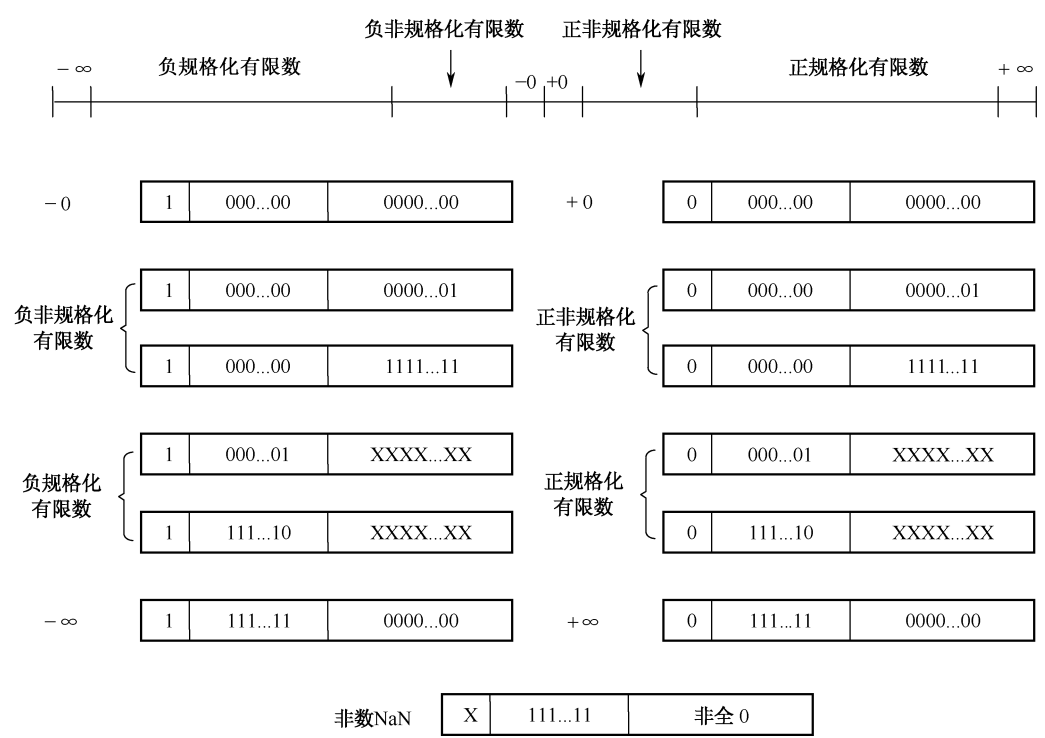


图 9-2 浮点数据类型

x87 FPU 除支持 IEEE 754 标准的 32 位单精度浮点格式和 64 位双精度浮点格式之外，还引入 80 位扩展精度浮点格式，它的最高位是 1 位符号，随后 15 位用于指数，最低 64 位是有效数字。扩展精度浮点数主要用于内部存储中间结果，以保证最终数值的精度。很多计算机中并没有 80 位扩展精度这种数据类型。x87 FPU 还支持 16、32 和 64 位的 3 种整型数据类型，以及 18 位十进制数的 BCD 码。

9.1.2 浮点寄存器

类似整数处理器，浮点处理单元也采用一些寄存器协助完成浮点操作。对程序员来说，组成 x87 FPU 浮点执行环境的寄存器主要是 8 个浮点数据寄存器和几个专用寄存器，它们是标记寄存器、状态寄存器和控制寄存器等。

1. 浮点数据寄存器

x87 FPU 浮点处理单元有 8 个浮点数据寄存器（FPU Data Register）：FPR0~FPR7，如图 9-3 所示。每个浮点寄存器都是 80 位的，以扩展精度格式存储数据。当其他类型数据压入数据寄存器时，自动转换成扩展精度；相反，数据寄存器的数据取出时，系统也会自动转换成要求的数据类型。x87 FPU 采用早期处理器的堆栈结构，8 个数据寄存器不采用随机存取，而是按照“后进先出”的堆栈原则工作，并且首尾循环。所以，浮点数据寄存器常被称为浮点数据栈，或浮点寄存器栈。

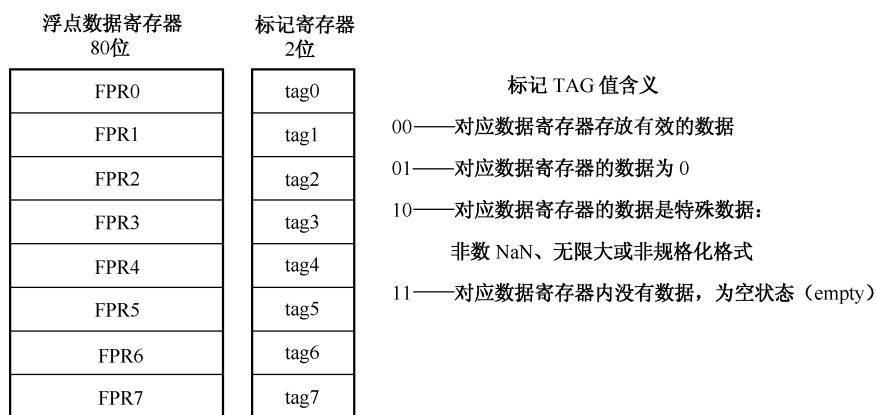


图 9-3 浮点数据寄存器

为了表明浮点数据寄存器中数据的性质，对应每个 FPR 寄存器，都有一个 2 位的标记（Tag）位，这 8 个标记 tag0～tag7 组成一个 16 位的标记寄存器。

2. 浮点状态寄存器

16 位浮点状态寄存器表明浮点处理单元当前的各种操作状态，每条浮点指令都对它进行修改以反映执行结果，其作用与整数处理单元的标记寄存器 EFLAGS 相当，如图 9-4（a）所示。

（1）堆栈标志

堆栈有栈顶，浮点状态寄存器的 TOP（D13～D11）字段指明哪个浮点数据寄存器 FPR 是当前栈顶。这 3 位组合的数字 0～7 指示当前栈顶的数据寄存器 FPR0～FPR7 的编号。

浮点数据寄存器栈可能出现溢出操作错误。当下一个数据寄存器已存有数据时（非空寄存器），继续压入数据就发生堆栈上溢（Stack Overflow）；当上一个浮点寄存器已没有数据时（空寄存器，标记位 tag=11B），继续取出数据就发生堆栈下溢（Stack Underflow）。SF（D6）堆栈溢出标志为 1，表示寄存器栈有溢出错误。条件码 C1 说明是堆栈上溢（C1=1）还是下溢（C1=0）。条件码（Condition Code）共 4 位，其他 3 位 C3/C2/C0 保存浮点比较指令的比较结果。

（2）异常标志

状态寄存器的低 6 位反映了浮点运算可能出现的 6 种异常。

- ⊙ PE 精度异常（Precision Exception）为 1，表示结果或操作数超出指定的精度范围，出现了不准确结果。
- ⊙ UE 下溢异常（Underflow Exception）为 1，表示非 0 的结果太小，出现下溢异常。
- ⊙ OE 上溢异常（Overflow Exception）为 1，表示结果太大，出现上溢异常。
- ⊙ ZE 被零除异常（Zero Divide Exception）为 1，表示出现除数为 0 的错误。
- ⊙ DE 非规格化操作数异常（Denormal Operand Exception）为 1，表示至少有一个操作数是非规格化的。
- ⊙ IE 非法操作异常（Invalid Operation Exception）为 1，表示操作为非法，如用负数开平方等。

除 DE 非规格化操作数异常外，IEEE 754 标准也定义了上述其他异常。另外，错误总结 ES（Error Summary）标志在任何一个未被屏蔽的异常发生时，都会置位。浮点处理单元忙位

B (FPU Busy) 为 1，表示浮点处理单元正在执行浮点指令；为 0，表示空闲。

3. 浮点控制寄存器

16 位浮点控制寄存器用于控制浮点处理单元的异常屏蔽、精度和舍入操作，如图 9-4 (b) 所示（下面一行数字是初始值）。

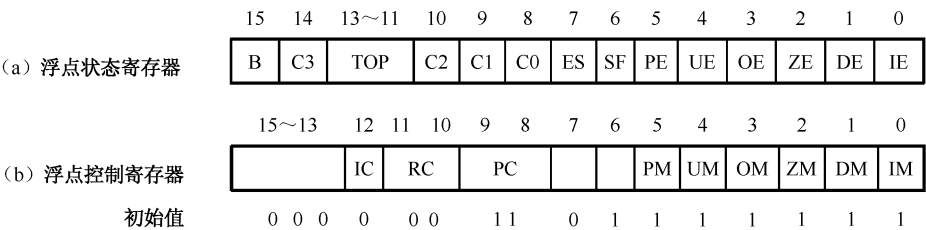


图 9-4 浮点状态和控制寄存器

(1) 异常屏蔽控制 (Mask Control)

控制寄存器的低 6 位决定 6 种错误是否被屏蔽，其中任意一位为 1 表示不允许产生相应异常（屏蔽），即精度异常屏蔽 PM (Precision Mask)、下溢异常屏蔽 UM (Underflow Mask)、上溢异常屏蔽 OM (Overflow Mask)、被零除异常屏蔽 ZM (Zero divide Mask)、非规格化异常屏蔽 DM (Denormal Operand Mask) 和非法操作异常屏蔽 IM (Invalid operation Mask)。

浮点程序设计中，异常的处理是一个难点，尤其异常处理程序的编写比较复杂。通过屏蔽特定异常，程序员可以将多数异常留给 FPU 处理，而主要处理严重的异常情况。x87 FPU 初始化后默认屏蔽所有异常。

(2) 精度控制 (Precision Control)

精度控制 PC 有 2 位，控制浮点计算结果的精度。PC=00，32 位单精度；PC=01，保留；PC=10，64 位双精度；PC=11，80 位扩展精度。

程序通常采用默认扩展精度，以使结果有效数最多，即精度最高。采用单精度和双精度是为了支持 IEEE 标准，也使得在用低精度数据类型进行计算时精度不变化。精度控制位仅仅影响浮点加、减、乘、除和平方指令的结果。

(3) 舍入控制 (Rounding Control)

只要可能，浮点处理单元就会按照要求格式（单、双或扩展精度）产生一个精确值。但是，经常出现精确值无法用要求的操作数格式编码的情况，这就需要进行舍入操作。2 位舍入控制 RC 控制浮点计算采用的舍入类型，如表 9-3 所示。

表 9-3 舍入控制

RC	舍入类型	舍入原则
00	就近舍入（偶）	舍入结果最接近准确值。如果上下两个值一样接近，就取偶数结果（最低位为 0）
01	向下舍入（趋向-∞）	舍入结果接近但不大于准确值
10	向上舍入（趋向+∞）	舍入结果接近但不小于准确值
11	向零舍入（趋向 0）	舍入结果接近但绝对值不大于准确值

“就近舍入” (Round to Nearest) 是默认的舍入方法，类似“四舍五入”原则，适合于大多数应用程序，它提供了最接近准确值的近似值。例如，有效数字超出规定数位的多余数字是 1001，它大于超出规定最低位的一半（即 0.5），故最低位进 1。如果多余数字是 0111，它

小于最低位的一半，则舍掉多余数字（截断尾数、截尾）。对于多余数字是 1000，正好是最低位一半的特殊情况，最低位为 0，则舍掉多余位，最低位为 1，则进位 1，使得最低位仍为 0（偶数）。

“向下舍入”（Round Down）用于得到运算结果的上界。对正数，就是截尾；对负数，只要多余位不全为 0 则最低位进 1。

“向上舍入”（Round Up）用于得到运算结果的下界。对负数，就是截尾；对正数，只要多余位不全为 0，则最低位进 1。

“向零舍入”（Round toward Zero）就是向数轴原点舍入，不论是正数还是负数都是截尾，使绝对值小于准确值，所以也称为截断舍入（Truncate）。它常用于浮点处理单元进行整数运算。

另外，无穷大控制 IC（Infinity Control）用于兼容 Intel 80287 数值协处理器。它对以后的 x87 FPU 没有意义。

【例 9-3】把实数 0.2 转换成浮点数据格式。

将实数“0.2”转换为二进制数，但它是“0011”的无限循环数据：

$$0.2 = 0.00110011\dot{00}11 \quad B = 1.100110011001100110011\dot{00}11 \quad B \times 2^{-3}$$

于是，符号位=0。

指数部分是-3，8 位阶码为 01111100（即 $-3+127=124$ ）。

有效数字是无限循环数，按照单精度要求取前 23 位是 10011001100110011001100，后面是 110011 B，需要进行舍入处理。按照默认的最近舍入方法，应该进位 1。所以，有效数字编码是 10011001100110011001101。

这样，0.2 表示成单精度浮点数为：

$$\begin{aligned} &0 \ 01111100 \ 10011001100110011001101 \ B \\ &= 0011 \ 1110 \ 0100 \ 1100 \ 1100 \ 1100 \ 1101 \ B \\ &= 3E4CCCCD \ H \end{aligned}$$

就是 3E4CCCCDH（参见例 9-4 程序的验证，也可以执行例 8-7 程序进行验证）。通过这个例子看到，计算机把一个简单的“0.2”都表达不准确，可见，浮点格式数据只能表达精度有限的近似值。但如果采用 BCD 码，真值“0.2”可以表达为“00000010B”（即 02H，假设小数点在中间）。

9.1.3 浮点指令编程

x87 FPU 具有自己的指令系统，共有几十种浮点指令，指令助记符均以 F 开头。浮点指令系统包括了常用的指令类型：浮点传送指令、浮点算术运算指令、浮点超越函数指令、浮点比较指令和 FPU 控制指令。

浮点指令一般需要 1 个或 2 个操作数，数据存于浮点数据寄存器或主存中（不能是立即数），主要有 3 种寻址方式：

- ◎ 隐含寻址——操作数在当前数据寄存器顶 ST(0)。许多浮点指令的一个隐含（目的）操作数是 ST(0)；汇编格式中 ST 等同于 ST(0)。这是堆栈结构下操作数的一个特点，常使得汇编语言程序员感到困惑，增加了编写浮点指令程序的难度。
- ◎ 寄存器寻址——操作数在指定的数据寄存器栈中 ST(i)。其中，i 是相对于当前栈顶 ST(0)而言，即 $i=0\sim 7$ 。

◎ 存储器寻址——操作数在主存中；主存中的数据可以采用任何存储器寻址方式。

1. 浮点传送指令

浮点数据传送指令完成主存与栈顶 ST(0)、数据寄存器 ST(*i*)与栈顶之间的浮点格式数据的传送。浮点数据寄存器是一个首尾相接的堆栈，所以它的数据传送实际上是对堆栈的操作，有些要改变堆栈指针 TOP，即修改当前浮点数据寄存器栈顶。

(1) 取数指令

取数指令 FLD 从存储器或浮点数据寄存器取得 (Load) 数据，压入 (Push) 寄存器栈顶 ST(0)。“压栈”的操作是：使栈顶指针 TOP 减 1，数据进入新的栈顶 ST(0)，见图 9-5 (a)。压栈操作改变了指针 TOP 指向的数据寄存器，即原来的 ST(0)成为现在的 ST(1)、原 ST(1)为现 ST(2) ……其他浮点指令实现数据进入寄存器栈都伴随有这个压栈操作。数据进入寄存器栈前由浮点处理单元自动转换成扩展精度浮点数。

(2) 存数指令

存数指令 FST 将浮点数据寄存器栈顶数据存入 (Store) 主存或另一个浮点数据寄存器，寄存器栈没有变化。数据取出后按要求格式自动转换，并在状态寄存器中设置相应异常标志。

(3) 存数且出栈指令

存数且出栈指令 FSTP 除执行相应存数指令功能外，还要弹出 (Pop) 栈顶。“出栈”的操作是：将栈顶 ST(0)清空 (使对应的标记位等于 11B)，并使 TOP 指针加 1，见图 9-5 (b)。出栈操作改变了指针 TOP 指向的数据寄存器，即原来的 ST(1)成为现在的 ST(0)，原 ST(2)为现 ST(1) ……浮点指令集中还有一些这样的执行“出栈”操作的指令，它们的指令助记符都是用 P 结尾。



图 9-5 浮点数据寄存器栈的操作

浮点数据传送指令有一组常数传送指令，它们将浮点运算过程中经常使用的常数按扩展精度压入寄存器栈顶 ST(0)。例如，传送 0、1、 π 和 $\log_2 10$ 等常数的浮点指令依次是 FLDZ、FLD1、FLDPI 和 FLDL2T。

浮点交换指令 FXCH 实现栈顶 ST(0)与任一个寄存器 ST(*i*)之间的数据交换。由于许多浮点指令只对栈顶操作，有了这个交换指令，就可以方便地对其他数据寄存器单元进行操作。

【例 9-4】 浮点传送程序。

```

.model small           ;基于模拟 DOS 平台
.686                  ;支持 32 位指令
.stack
.data
f32d  real4 100.25,0.2    ;单精度浮点数
f64d  real8 -0.2109375    ;双精度浮点数
f80d  real10 100.25e9      ;扩展精度浮点数
varf  real4 ?,?
i32d  dd 3e4ccccdh        ;0.2 的编码（参见例 9-3）
.code
.startup
finit                 ;初始化 FPU
fld f32d              ;压入单精度浮点数 f32d
fld f64d              ;压入双精度浮点数 f64d
fld f80d              ;压入扩展精度浮点数 f80d
fldpi                 ;压入 $\pi$ （3.1415926...）
fst varf              ;将栈顶数据 $\pi$ 传送到变量 VARF
fstp varf+4           ;将栈顶数据 $\pi$ 弹出到变量 VARF+4
mov eax,dword ptr f32d+4 ;取 0.2（二进制编码）
cmp eax,i32d          ;比较编码是否相同
jz dispy
mov dl,'N'             ;不相同，显示 N
jmp dispn
dispy: mov dl,'Y'       ;相同，显示 Y
dispn: mov ah,2
int 21h
.exit
end

```

本例程序基于模拟 DOS 平台，使用 32 位指令编写，用于演示浮点传送指令的功能，并验证“0.2”的编程。程序运行应显示“Y”，因为实数“0.2”的浮点格式编码是 3E4CCCCDH（参见例 9-3，也可以通过本例程序的列表文件查看），与其比较的结果当然应该相同了。

数据定义伪指令 DD（DWORD）、DQ（QWORD）和 DT（TBYTE）依次定义 32、64 和 80 位数据，可用于定义整数变量，也可用于定义单精度、双精度和扩展精度浮点数变量。实数表达至少有一个数字和一个小数点，如果没有小数点则是一个整数。为了相互区别，MASM 6.11 建议采用 REAL4、REAL8、REAL10 定义单、双、扩展精度浮点数，但不能出现纯整数（其实，整数后面补个小数点就可以了）。相应的数据属性依次是 DWORD、QWORD、TBYTE。另外，实常数可以用 E 表示 10 的幂。

每当执行一个新的浮点程序时，第一条指令都应该是初始化 FPU 的指令 FINIT。该指令清除浮点数据寄存器栈和异常，为程序提供一个“干净”的初始状态，否则遗留在浮点寄存器栈中的数据可能会产生堆栈溢出。另一方面，浮点指令程序段结束，最好清空浮点数据寄存器。

2. 其他浮点指令

浮点算术运算指令实现浮点数据的加 (FADD)、减 (FSUB)、乘 (FMUL)、除 (FDIV) 运算, 还包括求绝对值 (FABS)、求平方根 (FSQRT) 和取整 (FRNDINT) 等指令。

浮点超越函数指令实现对实数求三角函数、指数和对数等运算, 包含计算正切 (FPTAN)、反正切 (FPATAN)、正弦 (FSIN)、余弦 (FCOS)、正弦和余弦 (FSINCOS)、指数 (F2XM1)、对数 (FYL2X) 等指令。

浮点比较指令比较栈顶数据与指定的源操作数, 比较结果通过浮点状态寄存器反映, 如检查浮点数据类型 (FXAM)、与零比较 (FTST)、浮点数比较指令 (FCOM) 等指令。

FPU 控制指令用于控制和检测浮点处理单元的状态及操作方式, 如 FPU 初始化 (FINIT)、浮点空操作 (FNOP)、保存浮点状态 (FSAVE)、设置浮点状态 (FRSTOR) 等指令。

【例 9-5】 计算圆面积的程序。

利用嵌入汇编与 C++ 程序进行混合编程, 求圆面积。

```
#include <iostream.h>
float area(float radius);
int main()
{
    float ftemp;
    cout<<"请输入圆的半径: \t";
    cin>>ftemp;
    cout<<endl<<"该圆的面积是: \t"<<area(ftemp)<<endl;
    return 0;
}
float area(float radius)
{
    float ftemp;           // 定义局部变量, 用于返回值
    __asm {                // 嵌入式汇编代码部分
        fldpi               ; $\pi$  压入栈顶
        fld fradius         ;半径值  $R$  压入栈顶
        fmul st(0),st(0)    ;乘积:  $R \times R$ 
        fmul                ;求出面积:  $\pi \times R^2$ , 并出栈
        fstp ftemp          ;弹出面积:  $\pi R^2$ 
    }
    return(ftemp);
}
```

9.2 多媒体指令

计算机的传统应用领域是科学计算、信息处理和自动控制。随着个人计算机大量进入家庭, 人们希望在计算机中感受多彩的现实世界和虚幻的未来世界。计算机不仅要处理文字, 还要处理图形图像, 以及音频、动画和视频等多种媒体形式, 于是多媒体计算机在 20 世纪 90 年代初出现, 多媒体技术也应运而生。多媒体技术是将多媒体信息, 经计算机设备的获取、编辑、存储等处理后, 以多媒体形式表现出来的技术。为了满足多媒体技术对大量数据快速处理的需要, Intel 公司在其 Pentium 系列处理器中加入了多媒体指令。

多媒体指令的关键技术是采用了“单指令多数据 SIMD”(Single Instruction Multiple Data)结构,即利用一条多媒体指令能够同时处理多对数据,极大地提高了处理器性能。所以,多媒体指令也常称为 SIMD 指令。现在,多媒体指令已经广泛应用于高性能通用处理器和专用处理器(如数字信号处理器 DSP)中,并随着计算机、多媒体播放器和多功能手机等电子设备影响着我们的工作和生活。

9.2.1 MMX

MMX (MultiMedia eXtension) 意为多媒体扩展,是 1996 年 Intel 公司正式公布的处理器增强技术,针对多媒体信息处理中的数据特点,新增了 57 条多媒体指令,用于处理大量的整型数据。

1. MMX 数据类型

根据多媒体数据的特点,MMX 技术引入了“紧缩(Packed)整型数据”,以及基本的、通用的紧缩整型指令(MMX 指令、整型 SIMD 指令)。

紧缩整型数据是指将多个 8、16、32 或 64 位的整型数据组合形成一个整体。MMX 指令采用 64 位紧缩整型数据,可以表示 8 字节(Packed Byte)、4 个字(Packed Word)、2 个双字(Packed Doubleword)或 1 个 4 字(Packed Quadword),如图 9-6 所示。8 字节的紧缩数据按照小端方式连续存放在主存中。

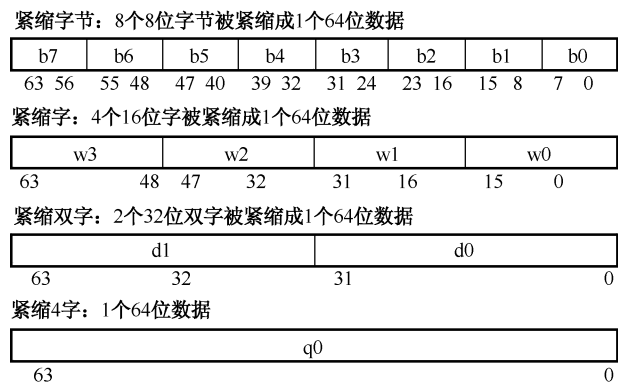


图 9-6 MMX 紧缩整型数据格式

2. MMX 寄存器

MMX 技术含有 8 个 64 位的 MMX 寄存器: MM0、MM1、……、MM7,用于对紧缩整型数据进行运算。

MMX 寄存器设计为随机存取、便于编程使用,但实际上是借用 8 个浮点数据寄存器实现的。x87 FPU 有 8 个浮点数据寄存器 FPR,以堆栈方式存取。每个浮点数据寄存器有 80 位,高 16 位用于指数和符号,低 64 位用于有效数字。MMX 利用其 64 位有效数字部分用做随机存取的 64 位 MMX 寄存器。Intel 通过将 MMX 寄存器影射到已有的浮点数据寄存器中,并未增加任何新的物理寄存器就得到了 8 个 MMX 寄存器,也没有增加任何状态标志等。这样就保持了与原 IA-32 处理器的软件兼容。

3. MMX 指令

MMX 指令是一组处理紧缩整型多媒体数据的通用指令，包括了一般整型指令的主要类型，可以分成如下几类：MMX 算术运算指令、MMX 比较指令、MMX 移位指令、MMX 类型转换指令、逻辑指令、传送指令和状态清除指令 EMMS。

MMX 指令的助记符除传送（MOVD 和 MOVQ）和清除指令（EMMS）外，都以 P 开头。另外，多数指令的助记符有一个说明数据类型的后缀：B、W、D 和 Q，依次表示紧缩字节、字、双字和四字。

MMX 的每种指令都是针对多媒体数据处理的需要精心设计的，大多数指令都体现了单指令多数据 SIMD 特性。例如，MMX 指令 PADD[B,W,D]实现两个紧缩数据的加法，如图 9-7 所示。PADDB 指令的操作数是 8 对互相独立的 8 位字节数据元素（紧缩字节类型），而 PADDW 指令的操作数是 4 对互相独立的 16 位字数据元素（紧缩字类型），PADDD 指令的操作数是 2 对互相独立的 32 位双字数据元素（紧缩双字类型）。各数据元素相加形成各自的结果，相互间没有关系和影响。在多媒体软件中大量存在着这种需要并行处理的数据。

	b07	b06	b05	b04	b03	b02	b01	b00
+	b17	b16	b15	b14	b13	b12	b11	b10
	b07+b17	b06+b16	b05+b15	b04+b14	b03+b13	b02+b12	b01+b11	b00+b10
	7F	FE	F0	00	00	03	12	34
+	00	03	30	00	FF	FE	43	21
	7F	01	20	00	FF	01	55	55

(a) PADDB指令

	w03	w02	w01	w00		7FFE	F000	0003	1234
+	w13	w12	w11	w10	+	0003	3000	FFFE	4321
	w03+w13	w02+w12	w01+w11	w00+w10		8001	2000	0001	5555

(b) PADDW指令

	d01	d00		7FFE000	00031234
+	d11	d10	+	00033000	FFFE4321
	d01+d11	d00+d10		80022000	00015555

(c) PADDD指令

图 9-7 MMX 加法 PADD[B,W,D]指令

4. 环绕运算和饱和运算

环绕（Wrap-around）运算就是通常的算术运算，是指当无符号数据的运算结果超过其数据类型界限时，它进行正常进位或借位运算。但是，MMX 技术没有新增任何标志，MMX 指令也并不影响状态标志，所以每个进位或借位并不能反映出来。图 9-7 演示的 PADD 指令就是环绕加法运算。

饱和（Saturation）运算是 MMX 指令的一个特点，是指运算结果超过其数据界限时，其结果被最大或最小值所替代。饱和运算有带符号数和无符号数之分，因为带符号和无符号数据的界限是不同的，如表 9-4 所示。

表 9-4 各数据类型上的、下界限

数据类型	无符号数据	有符号数据
字节	00H~FFH (255)	80H (−128)~7FH (127)
字	0000H~FFFFH (65535)	8000H (−32768)~7FFFH (32767)
双字	00000000H~FFFFFFFFH ($2^{32}-1$)	80000000H (− 2^{31})~7FFFFFFFH ($2^{31}-1$)

对无符号数据来说，有进位或借位就是超出范围，此时出现饱和。例如，无符号 16 位字的数据界限是 0000~FFFFH，则无符号饱和运算：

7FFE_H+0003_H=8001_H (不饱和)
0003_H+FFFE_H=FFFF_H (饱和)
7FFE_H−0003_H=7FFB_H (不饱和)
0003_H−FFFE_H=0000_H (饱和)

对有符号数据来说，产生溢出就是超出范围，此时出现饱和。例如，有符号 16 位字的数据界限是 8000H~7FFFH，则带符号饱和运算：

7FFE_H+0003_H=7FFF_H (饱和)
0003_H+FFFE_H=0001_H (不饱和)
7FFE_H−0003_H=7FFB_H (不饱和)
0003_H−FFFE_H=0005_H (不饱和)

饱和运算对许多图形处理程序非常重要。例如，图形正在进行黑色浓淡处理时，可以有效地防止由于环绕相加导致的黑色像素突变为白色，因为饱和运算将计算结果限制到最大的黑色值，绝不会溢出成白色。

5. 乘加指令

具有“乘-加”运算指令是 MMX 技术的又一个特点，因为进行数据相乘然后乘积求和是向量点积、矩阵乘法、快速傅里叶变换等主要的运算，而后者又是处理图像、音频和视频等多媒体数据的最基本算法。

MMX 乘加指令 PMADDWD 将源操作数的 4 个有符号字与目的操作数的 4 个有符号字分别相乘，产生 4 个有符号双字；然后，低位的 2 个双字相加并存入目的寄存器的低位双字，高位的 2 个双字相加并存入目的操作数的高位双字，如图 9-8 所示。PMADDWD 指令对源和目的操作数的所有 4 个字都为 8000H 情况，结果处理为 80000000H。

w03	w02	w01	w00	7FFE	F000	0003	1234
w13	w12	w11	w10	0003	3000	FFFE	4321
w03×w13+w02×w12				w01×w11+w00×w10			
FD017FFA				04C5F4AE			

图 9-8 乘加指令 PMADDWD

9.2.2 SSE

采用 MMX 指令的 Pentium II 处理器取得了极大的成功，推动了多媒体应用软件的发展，同时也对处理器能力提出了更高的要求。于是，Intel 公司针对互联网的应用需求，使用 MMX 指令集的关键技术“单指令多数据 SIMD”，在 1999 年 2 月推出了具有 SSE（数据流 SIMD 扩展，Streaming SIMD Extensions）指令集的 Pentium III 处理器。

数据流 SIMD 扩展技术基于原来的 IA-32 编程环境，主要提供了 8 个 128 位的 SIMD 浮

点数据寄存器 XMM0~XMM7，增加了 70 条指令的 SSE 指令集，用于支持 128 位紧缩单精度浮点数据。

1. 紧缩单精度浮点数据

SSE 技术支持的主要数据类型是紧缩单精度浮点操作数（Packed Single-precision Floating-point），是将 4 个互相独立的 32 位单精度（Single-Precision，SP）浮点数据组合在一个 128 位的数据中，如图 9-9 所示。32 位单精度数据格式符合 IEEE 754 标准。

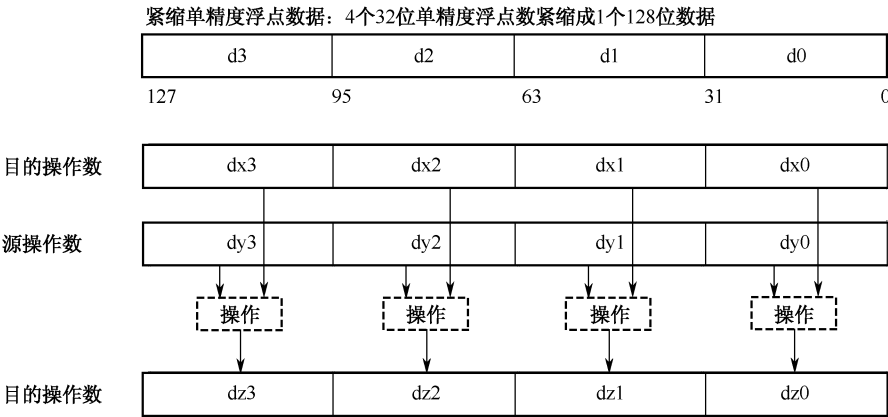


图 9-9 紧缩单精度浮点数据格式和 128 位操作模式

由于采用与紧缩整型数据类似的紧缩浮点数据，所以多数 SIMD 浮点指令一次可以处理 4 对 32 位单精度浮点数据。图 9-9 也演示了 SSE 指令支持的 128 位操作模式。

2. SSE 寄存器

SSE 技术提供了 8 个 128 位的 SIMD 浮点数据寄存器。每个 SIMD 浮点数据寄存器都可以直接存取，寄存器名为 XMM0~XMM7，用于存放数据而不能用于寻址存储器。

SSE 技术还提供了一个 32 位的控制状态寄存器 MXCSR(SIMD Floating-Point Control and Status Register)，相当于 x87 FPU 的浮点控制寄存器和浮点状态寄存器，用于屏蔽/允许数字异常处理程序、设置舍入类型、选择刷新至零模式、观察状态标志，如图 9-10 所示。

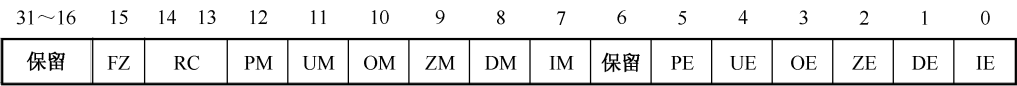


图 9-10 浮点 SIMD 控制/状态寄存器

MXCSR 寄存器的低 6 位是 6 个反映是否产生 SIMD 浮点无效数值异常的状态标志，其含义与 x87 FPU 浮点异常标志的含义相同。某个异常状态位为 1，表示发生了相应的浮点数值异常。MXCSR 中的 D12~D7 是 6 个对应 SIMD 浮点数值异常的屏蔽控制标志，也与 x87 FPU 浮点异常屏蔽标志作用相同。某异常屏蔽控制位置位，相应异常就被屏蔽。复位后 MXCSR 寄存器的内容是 1F80H，表示屏蔽所有异常。

MXCSR 中 RC 是两个舍入控制位，控制 SIMD 浮点数据操作的舍入原则，也具有就近舍入（00）、向下舍入（01）、向上舍入（10）和向零舍入（11）4 个类型。默认采用就近舍入类型。

MXCSR 中的刷新至零标志 FZ（Flush-to-Zero）为 1，将允许刷新至零模式：当运算结

果出现下溢时，将使结果刷新至零。IEEE 标准对于出现下溢情况是得到非规格化结果（逐渐下溢）。显然刷新至零模式不与 IEEE 标准兼容，但是它却以损失一点精度为代价，在经常出现下溢的应用程序中取得了较快的执行速度。复位后，FZ=0，关闭刷新至零模式。

3. SSE 指令

SSE 指令集有 70 条指令，其中有 12 条为增强和完善 MMX 指令集而新增加的 SIMD 整数指令（助记符仍以字母 P 开头）、8 条高速缓冲存储器优化处理指令，最主要的则是 50 条 SIMD 单精度浮点处理指令。

50 条 SSE 指令系统的 SIMD 浮点指令分成若干组，有数据传送、算术运算、逻辑运算、比较、数据转换、数据组合、状态管理指令。很多指令都体现了 SIMD 特性，如加（ADDPS）、减（SUBPS）、乘（MULPS）、除（DIVPS）指令可以实现 4 对单精度浮点数的算术运算。取最大值（MAXPS）和取最小值（MINPS）指令可以分别取得 4 对单精度浮点数各自的最大、最小值。求平方根（SQRTPS）、求倒数（RCPPS）和求平方根的倒数（RSQRTPS）指令则可以用一条指令获得 4 个单精度浮点数的平方根、倒数和平方根后的倒数。

9.2.3 SSE2

MMX 技术主要提供了并行处理整型数据的能力，SSE 技术主要提供的是单精度浮点数据的并行处理能力。2000 年 11 月，Intel 公司推出 Pentium 4 微处理器，又采用 SIMD 技术加入了 SSE2 指令，扩展了双精度浮点并行处理能力。SSE2 技术主要新增了紧缩双精度浮点数据类型和 76 条浮点 SIMD 指令，增强了紧缩整型数据类型和相应的 68 条整型 SIMD 指令。

1. 紧缩双精度浮点数据

SSE2 技术包括 IA-32 处理器原有的 32 位通用寄存器、64 位 MMX 寄存器、128 位 XMM 寄存器，还包括 32 位的标志寄存器 EFLAGS 和浮点状态/控制寄存器 MXCSR 等，但没有引入新的寄存器和指令执行状态。SSE2 主要利用 XMM 寄存器新增了一种 128 位紧缩双精度浮点数据和 4 种 128 位 SIMD 整型数据类型，见图 9-11。



图 9-11 SSE 的数据类型

紧缩双精度浮点数（Packed Double-precision Floating-point）由两个符合 IEEE 754 标准的 64 位双精度浮点数组成，紧缩成一个双 4 字数据。128 位紧缩整数（128-bit Packed Integer）可以包含 16 字节整数、8 个字整数、4 个双字整数或 2 个 4 字整数。SSE2 技术可进行两组双精度浮点数据或 64 位整数操作，还可以进行 4 组 32 位整数、8 组 16 位整数和 16 组 8 位整数操作。例如，对 8 位整数，SSE2 指令可以同时进行 16 对数据的运算，而普通整数指令只能进行一对数据运算。

2. SSE2 指令

SSE2 指令集包含 76 条双精度浮点数据 SIMD 指令，与 SSE 指令集非常相似，也分成多组，有 SSE2 的传送、算术运算、逻辑运算、比较、数据转换和数据组合指令，如加（ADDPD）、减（SUBPD）、乘（MULPD）、除（DIVPD）指令可以实现 2 对双精度浮点数的算术运算；取最大值（MAXPD）和取最小值（MINPD）指令可以分别取得 2 对双精度浮点数各自的最大、最小值；求平方根（SQRTPD）指令则可以用一条指令获得 2 个双精度浮点数的平方根。

SSE2 技术除具有 76 条双精度浮点指令外，在原来 MMX 和 SSE 技术基础上补充了 68 条 SIMD 扩展整数指令、高速缓存控制和指令排序指令，共用 144 条 SIMD 指令。

9.2.4 SSE3

2003 年，Intel 公司利用 90nm 工艺生产了新一代 Pentium 4 处理器。其中新增了 13 条 SSE3 指令，有 10 条用于完善 MMX、SSE 和 SSE2 指令，1 条用于 x87 FPU 编程中浮点数转换为整数的加速，另外 2 条用于加速线程的同步。SSE3 指令的编程环境没有改变，也没有引入新的数据结构或新的状态。

SSE3 指令支持的水平运算和对称运算很有特色。

1. 水平运算指令

大多数 SIMD 指令进行垂直操作，即两个紧缩操作数的同一个位置数据进行操作，结果也保存在该位置。水平运算指令进行水平操作，即在同一个紧缩操作数的连续位置数据进行加或减。SSE3 指令有单精度浮点水平加法（HADDPS）、减法（HSUBPS）和双精度浮点水平加法（HADDPD）、减法（HSUBPD）指令，如图 9-12 所示。

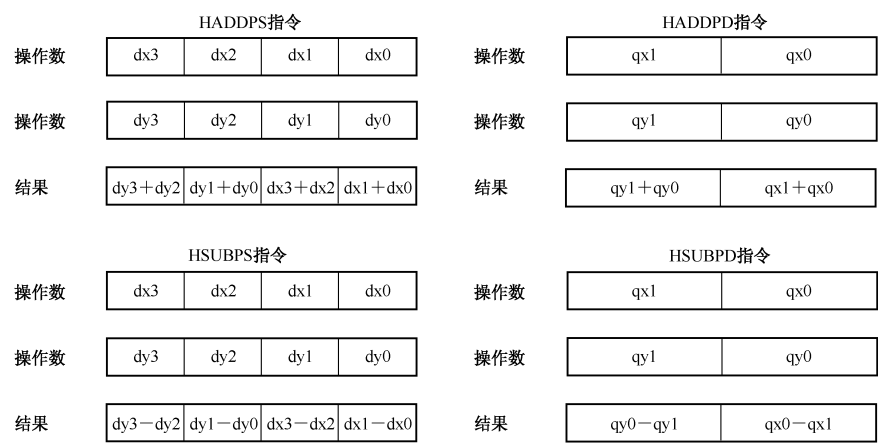


图 9-12 水平加法和减法指令

2. 对称加减指令

SSE3 的对称加减指令 ADDSUBPS 将第 2 和 4 个单精度浮点数对进行加法, 将第 1 个和第 3 个单精度浮点数对进行减法, 即对称处理。ADDSUBPD 指令对称处理的是双精度浮点数, 如图 9-13 所示。

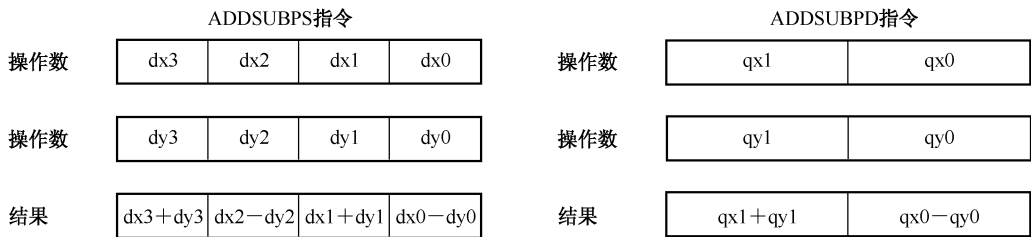


图 9-13 对称加减指令

3. SSSE3 指令

Intel Core 2 处理器引入补充 SSE3 指令, 即 SSSE3 指令 (Supplemental Streaming SIMD Extensions 3)。SSSE3 指令补充了 32 条指令, 包括 12 条水平运算指令, 6 条求绝对值指令, 2 条乘-加指令等。

9.3 64 位指令

2005 年, 在 PC 用户对 64 位技术的企盼和 AMD 公司兼容 32 位 80x86 的 64 位处理器的压力下, Intel 公司推出了扩展存储器 64 位技术 (Intel EM64T, Intel Extended Memory 64 Technology)。EM64T 技术是 IA-32 结构的 64 位扩展, 首先应用于支持超线程技术的 Pentium 4 至尊版 (支持双核技术) 和 6xx 系列 Pentium 4 处理器, 后来被称为 Intel 64 位结构 (Intel 64 Architecture)。随着 Intel 64 位技术的出现, IA-32 指令系统也扩展成为 64 位, 64 位软件也逐渐开始获得应用。

Intel 64 技术为软件提供了 64 位线性地址空间, 支持 40 位物理地址空间。Intel 64 技术在保护方式 (含虚拟 8086 方式)、实地址方式和系统管理 SMM 方式的基础上, 又引入了一个新的工作方式: 32 位扩展工作方式 (IA-32e)。

IA-32e 又有两个子工作方式:

① 兼容方式: 允许 64 位操作系统运行大多数 32 位软件而不需修改, 也可以运行大多数 16 位程序。虚拟 8086 方式和涉及硬件任务管理的程序不能在该方式下运行。

兼容方式由操作系统在代码段启动, 这意味着一个 64 位操作系统既可以在 64 位方式支持 64 位应用程序, 也可以在兼容方式支持 32 位应用程序 (不需进行 64 位编译)。

兼容方式类似 32 位保护方式。应用程序只能存取最低 4GB 地址空间, 使用 16 位或 32 位地址和操作数。

② 64 位方式: 允许 64 位操作系统运行存取 64 位地址空间的应用程序。

在 64 位工作方式, 应用程序还可以存取 8 个附加的通用寄存器、8 个附加的 SIMD 多媒体寄存器、64 位通用寄存器和 64 位指令指针等。

64 位方式引入了一个新的指令前缀 REX 用于存取 64 位寄存器和 64 位操作数。64 位方

式由操作系统在代码段启动，默认使用 64 位地址和 32 位操作数。默认操作数可以在每条指令的基础上用 REX 前缀超越，这样许多现有指令都可以使用 64 位寄存器和 64 位地址。

9.3.1 64 位方式的运行环境

64 位执行环境类似 32 位保护方式，不同之处在于，任务或程序可寻址 2^{64} 字节线性地址空间、 2^{40} 字节物理地址空间（软件可以访问 CPUID 指令获得处理器支持的实际物理地址范围）以及 64 位寄存器和操作数。

1. 64 位方式的寄存器

64 位方式新增 8 个 64 位通用寄存器 R8~R15。所以，64 位方式下有 16 个通用寄存器，默认是 32 位，用 EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP 和 R8D~R15D 表示。16 个通用寄存器还可以保存 64 位操作数，用 RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP 和 R8~R15 表示。它们也支持 16 位通用寄存器：AX, BX, CX, DX, SI, DI, SP, BP 或 R8W~R15W，还支持 8 位通用寄存器：使用 REX 前缀是 AL, BL, CL, DL, SIL, DIL, SPL, BPL 和 R8L~R15L；没有 REX 前缀是 AL, BL, CL, DL, AH, BH, CH, DH。

64 位方式为 SIMD 多媒体指令新增了 8 个 XMM 寄存器 XMM8~XMM15，现共有 16 个 128 位的 XMM 寄存器 XMM0~XMM15。

标志寄存器也扩展为 64 位，称之为 RFLAGS 寄存器。

2. 64 位方式的寻址方式

64 位方式的存储器操作数由一个段选择器和偏移地址访问，偏移地址可以是 16 位、32 位或 64 位。偏移地址通过如下部分组合（与 32 位存储器操作数寻址方式一样），其中基址和变址保存在 16 个通用寄存器之一：

- ⊙ 位移量——一个 8 位、16 位或 32 位数值
- ⊙ 基址——在 32 位或 64 位通用寄存器中的一个数值
- ⊙ 变址——在 32 位或 64 位通用寄存器中的一个数值
- ⊙ 比例因子——一个 2, 4 或 8 数值，用于乘以变址数值

64 位方式下，不管对应段描述符中的基地址是什么，都将 CS、DS、ES 和 SS 段寄存器的段基地址作为 0，这样为代码、数据和堆栈创建了一个平展的地址空间。FS 和 GS 是例外，它们可用于线性地址计算的附加基地址寄存器。

64 位方式的指令指针也是 64 位的，称为 RIP。所以，目标地址支持 RIP 相对寻址，使用有符号 32 位的位移量进行符号扩展成 64 位与下条指令的 RIP 计算有效地址。在 IA-32 兼容方式下，指令的相对寻址只能用于控制转移类指令；但在 64 位方式下，具有“mod reg r/m”寻址方式字段的指令也可以使用 RIP 相对寻址方式。

在 64 位方式，近指针（NEAR）是 64 位，即 64 位有效地址；所有近转移（CALL, RET, JCC, JCXZ, JMP 和 LOOP）的目的地址操作数都是 64 位。这些指令更新 64 位 RIP。

远指针（FAR）由一个 16 位段选择器和 16/32 位偏移地址（操作数为 32 位）或 64 位偏移地址（操作数为 64 位）组成。软件主要使用远转移改变特权层。因为立即数通常限制到 32 位，所以在 64 位方式，改变 64 位 RIP 的方法是使用间接转移寻址。也就是因为这个原因直接远转移寻址被删除。

9.3.2 64 位方式的指令

在 64 位方式，Intel 64 技术扩展了大多数整数通用指令的功能，使得它们都可以处理 64 位数据；但有一小部分整数通用指令不被 64 位方式所支持（但非 64 位方式仍然支持），还新增加了一些 64 位指令。

类似 IA-32 处理器 32 位指令系统对 Intel 80286 处理器 16 位指令系统的扩展，在 64 位方式，大多数通用指令可以处理 64 位操作数或者功能实现了向 64 位的自然增强。例如：

```
mov rax,r9
mov edx,[rsi+8]
mov qword ptr [ebx],rcx
mov eax,dword ptr [ecx*2+r10+0100h]
xchg r8,qvar           ;假设变量定义为“qvar dq 3456h”
xlatb                  ;AL=[RBX+AL]
lea r15,qvar           ;如果是 32 位地址则零位扩展成 64 位
mov rax,qword ptr varX ;对 64 位变量 varX 和 varY 的求和
add rax,qword ptr varY ;64 位操作数运算显然使用 64 位指令更加有效
add rax,rbx
sbb rdx,3721h
imul rbx               ;RDX:RAX←RAX×RBX
```

当然，64 位方式对有些指令并没有增加其 64 位处理能力，也就是没有改变，如输入 IN 和输出 OUT 指令，还有浮点 FPU 指令和 MMX 指令。SSE、SSE2 和 SSE3 指令可以利用 REX 前缀使用 8 个新增的 XMM 寄存器 XMM8-XMM15，如果它们的操作数是通用寄存器，则可以利用 REX.W 前缀访问 64 位通用寄存器。

还有些指令已经不被 64 位方式所支持，如 64 位方式不再支持 6 条十进制调整运算指令和边界检测指令 BOUND。单字节编码的 INC 和 DEC 指令因为与 16 个 REX 前缀代码一样，所以 64 位方式不被允许，但其他 INC 和 DEC 指令都是正常的。标志寄存器低字节传送指令 LAHF 和 SAHF 只有特定处理器支持。

1. 堆栈操作指令

64 位方式的堆栈指针 RSP 为 64 位，隐含使用 RSP 堆栈指针的指令（除远转移）默认采用 64 位操作数尺寸。所以，PUSH 和 POP 指令能将 64 位数据压入或弹出堆栈，但不能将 32 位数据的压入或弹出堆栈，使用 66H 操作数尺寸前缀可以支持 16 位数据的压入和弹出。将段寄存器内容压入 64 位堆栈时，指针自动调整成 64 位。64 位方式不支持 PUSHAX、PUSHAD、POPA 和 POPAD 指令。

PUSHF 和 POPF 指令在 64 位方式和非 64 位方式一样。PUSHFD 总是将 64 位 RFLAGS 压入堆栈（RF 和 VM 标志被清除）；POPDF 总是从堆栈弹出 64 位数据，然后将低 32 位进行零位扩展成 64 位存入 RFLAGS。

2. 符号扩展指令

零位扩展指令 MOVZX 现在支持将 8 或 32 位寄存器或存储器操作数进行零位扩展到 64 位通用寄存器。符号扩展指令 MOVSX 也支持 8 或 16 位寄存器或存储器操作数符号扩展到 64 位通用寄存器，但 32 位寄存器或存储器操作数符号扩展到 64 位通用寄存器使用 MOVSXD 指令。

例如，对于两个 32 位操作数求乘积，使用符号扩展成 64 位进行乘法更加有效：

```
movsxd rax,dword ptr varx
movsxd rcx,dword ptr vary
imul rax,rcx
```

在原有 CBW 和 CWDE 指令基础上，64 位方式新增有 CDQE 指令，后者将 EAX 数据符号扩展到 RAX。同样，CWD 和 CDQ 也扩展有新指令 CQO 指令，后者将 RAX 数据符号扩展到 RDX.RAX。

3. 串操作指令

在 64 位方式，串操作指令 MOVSB、CMPSB、SCASB、LODSB 和 STOSB，包括 INS 和 OUTSB 的源操作数用 RSI（使用 REX 前缀）或 DS:ESI 指示，目的操作数用 RDI（使用 REX 前缀）或 DS:EDI 指示。串操作的重复前缀使用 RCX（使用 REX 前缀）或 ECX。另外，64 位方式还增加了 4 条对 4 字数据的串操作指令：串比较 CMPSQ，串读取 LODSQ（4 字数据传送到 RAX），串传送 MOVSQ，串存储 STOSQ（将 RAX 的数据保存到主存）。

JCXZ 使用 CX 计数器，JECXZ 使用 ECX 计数器，新增 JRCXZ 指令使用 64 位 RCX 计数器。但是 LOOP、LOOPZ 和 LOOPNZ 指令在 64 位方式还可以使用 64 位寄存器 RCX 进行计数。它们仍然是短转移。

64 位方式的指令还有其他一些改变，不再详细介绍。Microsoft Visual C++.NET 2005 开始支持 Intel 64 技术，宏汇编程序 MASM 除保留有汇编 16 和 32 位指令的 ML.EXE 程序外，又为 64 位指令系统（称为 x64 结构）提供了 ML64.EXE 程序。ML64.EXE 程序能将 x64 结构的汇编语言源程序汇编成 x64 结构的目标代码 OBJ 文件。注意，Visual C++ 2005 文档中，Intel 64-bit 是指 Itanium（安腾）处理器的 64 位指令系统，AMD 64-bit 是指 AMD 公司的 64 位处理器指令系统。

习 题 9

9.1 简答题

- (1) 浮点数据为什么要采用规格化形式？
- (2) IEEE 标准规定单精度和双精度数据的有效数字位数分别为 23 和 52，但为什么说它们具有二进制 24 位和 53 位的精度？
- (3) 为什么有时使用非规格化浮点数据格式？
- (4) 什么是浮点格式中的 NaN？
- (5) 为什么浮点数据编码有舍入问题，而整数编码却没有？
- (6) 什么是就近舍入？
- (7) 多媒体指令为什么常被称为 SIMD 指令？
- (8) 为了支持 MMX 指令，处理器增加了标志和寄存器吗？
- (9) SSE3 指令的水平加减运算有什么特色？
- (10) 原来 8086 的所有指令都可以在 Intel 64 位工作方式下使用吗？

9.2 判断题

- (1) 浮点数据格式不能表达整数。
- (2) 一个 32 位数据是全 0，不管它是整数编码还是单精度浮点编码，都表示真值 0。

- (3) x87 FPU 有 8 个 80 位浮点数据寄存器, 可以随机存取。
- (4) 浮点指令中浮点寄存器常表达为 ST(*i*), 如 ST(0)总是对应 FPR0 寄存器。
- (5) x87 FPU 的指令系统只有浮点算术运算指令, 不支持复杂的求三角函数、指数和对数等运算。
- (6) 浮点指令也可以访问主存单元。
- (7) MMX 技术还不能支持浮点数据格式的处理。
- (8) SSE2 技术支持 128 位数据, 可表示 2 个双精度浮点数, 也支持 16 个 8 位整数。
- (9) IA-32 指令集结构升级为 64 位, 被称为 IA-64 指令集结构?
- (10) Intel 64 结构支持 16 个 64 位整数通用寄存器。

9.3 填空题

- (1) 对真值-125, 用补码表示是_____ ; 标准偏移码与补码只有一位不同, 所以是_____ ; 而浮点阶码则再减 1, 是_____。
- (2) 单精度浮点数据格式共有_____位, 其中符号位占一位, 阶码部分占_____位, 尾数部分有_____位。
- (3) 通过例 9-2 知道实数“100.25”的浮点格式编码是 42C88000H, 则“-100.25”的浮点格式编码是_____。
- (4) 单精度浮点规格化格式能表达的数据范围是从_____到_____。出现比最小数还要小的数据, 就是出现了_____ ; 出现比最大数还要大的数据, 就是出现了_____。
- (5) 如果要表达的实数大于浮点格式所能表达的最大数, 则 IEEE 754 标准将其编码称为_____, 特点是其阶码的编码均为_____, 有效数字的编码均为_____。
- (6) 为了扩大精度, x87 FPU 支持扩展精度格式, 共有_____位, 所以其浮点寄存器也设计为_____位。
- (7) IA-32 处理器的浮点指令助记符均以_____开头, 如浮点加减乘除指令的助记符依次是_____。
- (8) 对于有符号字节数据 78H 和 56H, 如果采用环绕加运算, 其和为_____ ; 若用饱和加运算, 则其和是_____。
- (9) MMX 技术引入 8 个_____位 MMX 寄存器, 名字是_____。SSE 技术又引入 8 个_____位 SIMD 浮点数据寄存器, 名字是_____。
- (10) Intel 64 结构新增 8 个通用寄存器, 名称为_____ ; 原来的 8 个通用寄存器被扩展为 64 位, 名称为_____。

9.4 已知 BF600000H 是一个单精度规格化浮点格式数据, 它表达的实数是什么?

9.5 实数真值 28.75 如果用单精度规格化浮点数据格式表达, 其编码是什么?

9.6 编程显示单精度浮点数据的编码(十六进制形式), 例如用于验证上一个习题结果。实数可以定义在数据段中。

9.7 解释如下浮点格式数据的有关概念:

- (1) 数据上溢和数据下溢
- (2) 规格化有限数和非规格化有限数
- (3) NaN 和无穷大

9.8 什么是紧缩整型数据和紧缩浮点数据? 扩展有 SSE3 指令的 Pentium 4 支持哪些紧

缩数据类型？

9.9 SIMD 是什么？举例说明 MMX 指令如何利用这个结构特点？

9.10 什么是环绕运算和饱和运算。给出如下结果：

- (1) 环绕加： $7F38H + 1707H$
- (2) 环绕减： $1707H - 7F38H$
- (3) 无符号饱和加： $7F38H + 1707H$
- (4) 无符号饱和减： $1707H - 7F38H$
- (5) 有符号饱和加： $7F38H + 1707H$
- (6) 有符号饱和减： $1707H - 7F38H$

附录 A 调试程序 DEBUG

DEBUG.EXE 是 DOS 提供的汇编语言级的可执行程序调试工具。

A.1 DEBUG 程序的调用

在 DOS 的提示符下，可输入 DEBUG 启动调试程序（如果在 Windows 图形界面下，需要首先进入模拟 DOS 环境或者控制台窗口）：

`DEBUG [路径\文件名][参数 1][参数 2]`

DEBUG 后可以不带文件名，仅运行 DEBUG 程序；需要时，再用 N 和 L 命令调入被调试程序。命令中可以带有被调试程序的文件名，则运行 DEBUG 的同时，还将指定的程序调入主存；参数 1 和参数 2 是被调试的可执行程序所需要的参数。

在 DEBUG 程序调入后，根据有无被调试程序及其类型相应设置寄存器组的内容，发出 DEBUG 的提示符“—”，此时就可用 DEBUG 命令来调试程序。

运行 DEBUG 程序时，如果不带被调试程序，则所有段寄存器值相等，都指向当前可用的主存段；除 SP 之外的通用寄存器都设置为 0，而 SP 指示当前堆栈项在这个段的尾部；IP=0100H；状态标志都是清 0 状态。

运行 DEBUG 程序时，如果带入的被调试程序扩展名不是 .EXE，则 BX 和 CX 包含被调试文件大小的字节数（BX 为高 16 位），其他同不带被调试程序的情况。

运行 DEBUG 程序时，如果带入的被调试程序扩展名是 .EXE，则需要重新定位。此时，CS:IP 和 SS:SP 根据被调试程序确定，分别指向代码段和堆栈段。DS=ES 指向当前可用的主存段，BX 和 CX 包含被调试文件大小的字节数（BX 为高 16 位），其他通用寄存器为 0，状态标志都是清 0 状态。

A.2 DEBUG 程序的命令

DEBUG 调试程序只能通过执行命令实现调试，结果随后显示。命令虽然简单，却是最基本的方法，微软其他调试程序都继承了这些基本命令。

DEBUG 的命令都是一个字母，后跟一个或多个参数：字母 [参数]。

使用命令时，需要注意：

- ① 字母不分大小写。
- ② 只使用十六进制数，没有后缀字母。
- ③ 分隔符（空格或逗号）只在两个数值之间是必须的，命令和参数间可无分隔符。
- ④ 每个命令只有按了回车键后才有效，可以用 Ctrl+Break 中止命令的执行。
- ⑤ 命令如果不符合规则，则将以“error”提示，并用“^”指示错误位置。

许多命令的参数是主存逻辑地址，形式是“段基地址：偏移地址”。其中，段基地址可以是段寄存器名或数值，偏移地址是数值。如果不输入段地址，则采用默认值，可以是默认段寄存器值。如果没有提供偏移地址，则通常就是当前偏移地址。

对主存操作的命令还支持地址范围这种参数，它的形式是“开始地址 结束地址”（结束地址不能具有段地址），或者是“开始地址 L 字节长度”。

1. 显示命令 D

D (Dump) 命令显示主存单元的内容，其格式如下（注意分号后的部分用于解释命令功能，不是命令本身，下同）：

D [地址] ; 显示当前或指定开始地址的主存内容

D [范围] ; 显示指定范围的主存内容

例如，显示当前（接着上一个 D 命令显示的最后一个地址）主存内容的一个示例（命令前的短画线是提示符不需输入，下同）：

```
-d
1492:0100 41 EB EA 5E E3 0B F7 C2-01 00 74 1C 80 3C 2E 74 A..^.....t.<.t
1492:0110 47 83 3E 75 E0 02 75 0A-80 3E 7C E1 34 00 81 14 G.>u..u..>|.4...
```

显示内容的左边部分是主存逻辑地址，中间是连续 16 个字节的主存内容（十六进制数，以字节为单位），右边部分是这 16 字节内容的 ASCII 字符显示，不可显示字符用点“.”表示。一个 D 命令仅显示“8 行×16 字节”（80 列显示模式）内容。

再如：

-d 100 ; 显示数据段 100h 开始的主存单元

-d cs:0 ; 显示代码段的主存内容

-d2f0 L20 ; 显示 ds:2f0h 开始的 20h 个主存数据

2. 修改命令 E

E (Enter) 命令用于修改主存内容，它有两种格式：

E 地址 ; 格式 1，修改指定地址的内容

E 地址 数据表 ; 格式 2，用数据表的数据修改指定地址的内容

格式 1 是逐个单元相继修改的方法。例如，输入“e ds:100”，DEBUG 显示原来内容，用户可以直接输入新数据，然后按空格键显示下一个单元的内容，或者按“—”键显示上一个单元的内容；不需要修改可以直接按空格或“—”键。这样，用户可以不断修改相继单元的内容，直到用回车键结束该命令为止。

格式 2 可以一次修改多个单元，例如：

-e ds:100 F3`X`Y`Z`8D ; 用 F3`X`Y`Z`8D 这 5 个数据替代 DS:0100 ~ 0104 的原来内容

3. 填充命令 F

F (Fill) 命令用于对一个主存区域填写内容，同时改写原来的内容，其格式为：

F 范围 数据表

该命令用数据表的数据写入指定范围的主存。如果数据个数超过指定的范围，则忽略多出的项；如果数据个数小于指定的范围，则重复使用这些数据，直到填满指定范围。

4. 寄存器命令 R

R (Register) 命令用于显示和修改处理器的寄存器，它有 3 种格式。

R ; 格式 1，显示所有寄存器内容和标志位状态

例如，当刚进入 DEBUG 时，就可以执行该命令，显示示例如下：

AX=0000 BX=0000 CX=010A DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=18E4 ES=18E4 SS=18E4 CS=18E4 IP=0100 NV UP DI PL NZ NA PO NC
18E4:0100 C70604023801 MOV WORD PTR[0204],0138 DS:0204=0000

其中，前两行给出所有寄存器的值，包括各个标志状态。最后一行给出了当前 CS：IP 处的指令；由于这是一个涉及数据的指令，这一行的最后还给出相应单元的内容。

R 寄存器名；格式 2，显示和修改指定寄存器

例如，输入“r ax”，DEBUG 给出当前 AX 内容，冒号后用于输入新数据，如不修改，则按 Enter 键。

RF；格式 3，显示和修改标志位

DEBUG 将显示当前各个标志位的状态。显示的符号及其状态如表 A-1 所示，用户只要输入这些符号（来自英文缩写）就可以修改对应的标志状态，键入的顺序可以任意。

表 A-1 标志状态的表示符号

标 志	置位符号	复位符号
溢出 OF	OV (overflow)	NV (no overflow)
方向 DF	DN (down)	UP (up)
中断 IF	EI (enable interrupt)	DI (disable interrupt)
符号 SF	NG (negative)	PL (plus)
零位 ZF	ZR (zero)	NZ (no zero)
辅助 AF	AC (auxiliary carry)	NA (no auxiliary)
奇偶 PF	PE (parity even)	PO (parity odd)
进位 CF	CY (carry)	NC (no carry)

5. 汇编命令 A

A (Assemble) 命令用于将后续输入的汇编语言指令翻译成机器代码，其格式如下：

A [地址]；从指定地址开始汇编指令

A 命令中如果没有指定地址，则接着上一个 A 命令的最后一个单元开始；若还没有使用过 A 命令，则从当前 CS：IP 开始。

输入 A 命令后，就可以输入 8086 和 8087 指令，DEBUG 将它们汇编成机器代码，相继存放在指定地址开始的存储单元中，记住最后要输入一个回车结束 A 命令。

进行汇编的步骤如下：

- (1) 输入汇编命令 A [地址]，按回车，DEBUG 提示地址，等待输入汇编语言指令。
- (2) 输入汇编语言指令，按回车。
- (3) 如上继续输入汇编语言指令，直到输入所有指令。
- (4) 不输入内容就按回车，结束汇编，返回 DEBUG 的提示符状态。

A 命令支持标准的 8086（和 8087 浮点）指令系统以及汇编语言语句基本格式，但要注意以下一些例外：

- ⊙ 所有输入的数值都是十六进制数，不要有后缀字母。
- ⊙ 段超越指令需要在相应指令前，单独一行输入。
- ⊙ 段间（远）返回的助记符要使用 RETF。
- ⊙ A 命令也支持最常用的两个伪指令 DB 和 DW。

6. 反汇编命令 U

U (Unassemble) 命令将指定地址的内容按 8086 和 8087 机器代码翻译成汇编语言指令，其格式如下：

U [地址] ; 从指定地址开始，反汇编 32 个字节 (80 列显示模式)
U 范围 ; 对指定范围的主存内容进行反汇编

U 命令中如果没有指定地址，则接着上一个 U 命令的最后一个单元开始；若还没有使用过 U 命令，则从当前 CS : IP 开始。例如：

```
-u
14C8:0000 B8CD14    MOV AX,14CD
14C8:0003 8ED8      MOV DS,AX
14C8:0005 BA0600    MOV DX,0006
14C8:0008 B409      MOV AH,09
14C8:000A CD21      INT 21
```

屏幕显示的左边是主存逻辑地址，中间是该指令的机器代码，右边则是对应的指令汇编语言格式。

7. 运行命令 G

G (Go) 命令从指定地址开始执行指令，直到遇到断点或者程序结束返回操作系统。

G [=地址] [断点地址 1,断点地址 2,...,断点地址 10]

G 命令等号后的地址指定程序段运行的起始地址，如不指定则从当前的 CS : IP 开始运行。断点地址如果只有偏移地址，则默认是代码段 CS；断点可以没有，但最多只能有 10 个。

G 命令输入后，即开始运行程序，遇到断点（实际上就是断点中断指令 INT 3，其指令代码只有一个字节，是 CCH），停止执行，并显示当前所有寄存器和标志位的内容以及下一条将要执行的指令（显示内容同 R 命令），以便观察程序运行到此的情况。程序正常结束，将显示 “Program terminated normally”。

注意，G 命令以及后面的 T 和 P 命令要用等号 “=” 指定开始地址（未指定则是当前 CS : IP 地址）；并且该地址处应该保存有正确的指令代码序列，否则会出现不可预测的结果，例如 “死机”。

8. 跟踪命令 T

T (Trace) 命令从指定地址开始执行一条或数值参数指定条数的指令后停下来，如未指定地址则从当前的 CS : IP 开始执行。注意给出的执行地址前有一个等号，否则会被认为是被跟踪指令的条数（数值）。

T [=地址] ; 逐条指令跟踪
T [=地址] [数值] ; 多条指令跟踪

T 命令执行每条指令后都要显示所有寄存器和标志位的值（以及下一条指令）。

T 命令提供了一种逐条指令运行程序的方法，因此它也常被称为单步命令。T 命令可以利用处理器的单步中断，使程序员能够细致地观察程序的执行情况。T 命令逐条指令执行程序，遇到子程序 (CALL)、中断调用 (INT n) 指令以及循环体也不例外，也会进入到子程序、中断服务程序以及循环体当中执行。

9. 继续命令 P

P (Proceed) 命令类似 T 命令，只是不会进入子程序或中断服务程序中，遇到循环指令则一并执行完所有循环。当不需要调试子程序、中断服务程序以及循环体时，要应用 P 命令，而不是 T 命令。

P [=地址] ; 逐条指令跟踪
P [=地址] [数值] ; 多条指令跟踪

10. 退出命令 Q

Q (Quit) 命令使 DEBUG 程序退出，返回 DOS。Q 命令没有参数，也没有将主存中程序保存成磁盘文件的功能（可使用 W 命令保存）。

11. 命名命令 N

N (Name) 命令定义要保存磁盘文件的文件名，格式如下：

N 文件标识符 1[,文件标识符 2]

文件标识符是包含路径的文件全名。有了文件标识符，才可以用 L 或 W 命令把文件装载到主存或者把主存内容保存到磁盘。

12. 装载命令 L

L (Load) 命令把磁盘文件或扇区装载到主存以便进行调试，有两种装载格式。

L [地址] ; 格式 1: 装入由 N 命令指定的文件

格式 1 的 L 命令装载一个文件到指定的主存地址处；如未指定地址，则装入 CS : 100H 开始的存储区；对于 COM 和 EXE 文件，则一定装入 CS : 100H 位置处。

L 地址 驱动器 扇区号 扇区数 ; 格式 2: 装入指定磁盘扇区范围的内容

格式 2 的 L 命令装载磁盘的若干扇区（最多 80H=128）到指定的主存地址处；默认段地址是 CS。其中，0 表示 A 盘，1 表示 B 盘，2 表示 C 盘，……例如，将硬盘 C 分区的 DOS 引导扇区（逻辑扇区号为 0 的一个扇区）内容装入，然后查看的命令是：

```
-l 0 2 0 1  
-d cs:0
```

13. 写盘命令 W

W (Write) 把主存内容保存到磁盘，有两种写盘格式。

W [地址] ; 格式 1: 把数据写入由 N 命令指定的磁盘文件

格式 1 的 W 命令将指定开始地址的数据写入一个文件（这个文件应该已经用 N 命令命名）；如未指定地址则从 CS : 100H 开始。写入文件的字节数要事先存入 BX（高字）和 CX（低字）寄存器中。如果采用这个 W 命令保存可执行程序，扩展名应是 COM；它不能写入具有 EXE 和 HEX 扩展名的文件。

W 地址 驱动器 扇区号 扇区数 ; 格式 2: 把数据写入指定磁盘扇区范围

格式 2 的 W 命令将指定地址的数据写入磁盘的若干扇区（最多 80H）；如果没有给出段地址，则默认是 CS。其他说明同 L 命令。由于格式 2 的 W 命令直接对磁盘扇区写入，没有经过 DOS 文件系统管理，所以一定要小心，否则可能无法利用 DOS 文件系统读写该部分内容。

14. 其他命令

DEBUG 还有一些其他命令，简单罗列如下：

(1) 比较命令 C (Compare)

C 范围 地址 ; 将指定范围的内容与指定地址内容比较

(2) 十六进制数计算命令 H (Hex)

H 数字 1, 数字 2 ; 同时计算两个十六进制数字的和与差

(3) 输入命令 I (Input)

I 端口地址 ; 从指定 I/O 端口输入 1 字节，并显示

(4) 输出命令 O (Output)

O 端口地址 字节数据 ; 将数据输出到指定的 I/O 端口

(5) 传送命令 M (Move)

M 范围 地址 ; 将指定范围的内容传送到指定地址处

(6) 查找命令 S (Search)

S 范围 数据 ; 在指定范围内查找指定的数据

A.3 DEBUG 程序的使用

掌握调试程序的使用，需要经过上机实践。下面介绍一般的调试步骤，书中正文给出结合例题的具体调试过程。建议通过第 2 章数据寻址初步了解调试程序，通过第 3 章掌握程序片段的调试方法，通过第 4 章学习完整程序的调试方法，而通过第 5 章则可以灵活使用调试程序有重点的分析解决问题。

1. 程序片段的调试步骤

调试程序 DEBUG 特别适理解指令和程序片段的功能，基本步骤如下：

(1) 进入 DOS 环境（或 Windows 控制台）。

(2) 在 DOS 提示符下，输入 DEBUG，启动调试程序。

(3) 利用汇编命令 A，输入汇编语言指令序列，最后用回车结束。

(4) 利用反汇编命令 U，观察录入的指令序列是否正确。

(5) 设置必要的参数，例如寄存器值、存储单元内容。

(6) 利用跟踪命令 T、单步执行指令，并观察每条指令执行情况，如结果、标志等。如果含有不需调试的子程序、系统功能调用，则利用继续命令 P，单步执行指令，并观察每条指令执行情况，如结果、标志等。也可以设置断点，利用运行命令 G 执行指令序列，并检查程序段的执行结果与自己判断的结果是否相同；找出不相同的原因。

2. 可执行程序文件的调试步骤

经汇编连接生成的可执行文件，可以载入调试程序中进行运行、调试，观察运行结果是否正确、帮助排错等。一般的调试步骤如下：

(1) 进入 DOS 环境，带被调试文件启动调试程序。

(2) 执行寄存器命令 R，观察程序进入主存的情况。

(3) 从第一条执行指令位置开始，执行反汇编命令 U，显示程序指令及地址。

(4) 执行显示命令 D 了解变量等在数据段存储的初值等。

(5) 选择断点, 使用运行命令 G 执行到断点位置暂停, 观察运行结果。通常第一个断点应该是完成初始化功能的位置, 如 “.STARTUP” 语句结束 (使用 8086 指令的小型存储模型时, 通常的偏移地址为 0017H)。

(6) 根据需要单步或者断点调试重点的程序片段, 仔细观察寄存器值, 主存单元内容, 判断运行结果是否正确。单步调试时注意跟踪命令 T (进入子程序、循环体使用) 和继续命令 P (不调试子程序、循环体以及进行 DOS 功能调用时使用) 的区别, 有针对性地正确使用。

(7) 通常要在程序返回 DOS 前设置最后一个断点, 便于观察程序运行的最终结果。

3. 使用 DEBUG 的注意事项

缘于 DOS 命令行的操作方式和调试程序本身的功能所限, 使用 DEBUG 调试程序常会遇到如下一些问题, 请予以注意。

(1) 汇编命令 A 下的指令格式与 MASM 有一些区别。

调试程序 DEBUG 的汇编命令 A 所支持的汇编语言指令格式基本与 MASM 相同, 但有一些区别, 请注意一下规则:

- ⊙ 所有输入的数值都是十六进制, 没有后缀字母;
- ⊙ 段超越指令需要在相应指令前, 单独一行输入;
- ⊙ 支持基本的伪指令 DB、DW 和操作符 WORD PTR、BYTE PTR。

(2) 调试程序 DEBUG 中不支持标号。

例如输入 “again: cmp al,[bx]” 时, 标号 again 并不能输入, 只输入 “cmp al,[bx]”, 但注意一下该指令的偏移地址。当输入使用该标号的指令, 如 “loop again” 时, 标号 again 使用该指令所在的偏移地址。

但当输入指令, 如 “jnz minus” 时, minus 所在的偏移地址还不知道, 则可以暂时添上其本身的偏移地址。等到输入具有 minus 标号的指令时记住该指令的偏移地址。最后, 重新在 jnz minus 的地址处汇编该指令, 此时添入该标号 minus 所在指令的偏移地址。

(3) 避免非正常读写和执行。

进入调试程序之后, DOS 操作系统下的信息都可以读写访问, 所以使用 DEBUG 进行程序调试时要小心, 不然很容易导致系统死机。例如:

- ⊙ 执行 T、P 和 G 命令时, 要在等号后输入正确的地址, 尤其是调试一段程序后, 往往不再是默认的地址了;
- ⊙ 注意留心一下 CS 和 DS 值是否是该程序段的位置, 否则不能执行正确的代码和观察到正确的数据段内容;
- ⊙ G 命令要有断点地址 (或者程序段最后有断点中断指令 INT 3);
- ⊙ 含有系统功能调用的程序段要用 P 命令单步执行 (不进入功能调用本身的程序中)。
- ⊙ 调试程序仅是一个模拟的执行环境, 不是所有程序段都能运行。例如, 改变 TF 标志的程序段不能执行, 因为调试程序本身也要使用 TF。再如, 向外设端口输出内容的程序段也不应随意执行, 因为有可能导致该端口对应的外设非正常运行。

(4) 在 DEBUG 中调试的程序片段也可以保存, 以便以后使用或重复调试。

基本步骤如下:

- ⊙ 利用 A 命令输入符合 DEBUG 要求的程序段 (最后最好加上一条 INT 3 指令)。
- ⊙ 利用 R 命令在 BX 和 CX 中存入程序段的长度 (字节数)。注意, BX 为高 16 位, 通

常应该为 0，因为程序段长度通常不会超过 64KB。

- ⊙ 利用 N 命令为文件起名，注意文件扩展名只能是 COM。例如：

- a filename.com

- ⊙ 利用 W 命令保存，例如：

- W 程序段起始偏移地址

- ⊙ 利用 Q 命令退出 DEBUG 调试程序。

附录 B 常用 DOS 功能调用

表 B-1 列出了基本的键盘输入、显示输出等 DOS 功能调用（INT 21H），本书主要使用了 1、2、9 和 4CH 号功能（具体的应用参看正文），完整的 DOS 功能需要查看参考资源。

表 B-1 指令符号说明

功 能 号	参数及功能说明
AH=01H	出口参数：AL=ASCII 字符 功能说明：键盘输入一个字符。如果 AL=00H，应再次调用该功能获取扩展 ASCII 字符代码。本功能将在屏幕上显示输入的内容（有回显）。调用该功能如果没有按键输入，则一直循环等待直到按键才结束功能调用，此时控制返回调用程序
AH=02H	入口参数：DL=欲显示的 ASCII 字符 功能说明：在屏幕当前光标的位置显示一个字符
AH=06H	入口参数：DL=FFH（对应输入功能），DL=欲显示的 ASCII 字符（对应输出功能） 出口参数：AL=ASCII 字符 功能说明：对应入口参数 DL=FFH，是一个键盘输入功能调用。调用该功能不论是否有按键输入，都将结束功能调用。标志 ZF 反映是否按键：条件转移指令的为零条件 Z（ZF=1）表示无按键；不为零条件 NZ（ZF=0）表示有按键，AL 返回输入字符的 ASCII 码。如果 AL=00H，应再次调用该功能获取扩展 ASCII 字符代码。本功能不在屏幕上显示输入的内容（无回显） 对应入口参数 DL=ASCII 字符，是一个显示输出功能调用，与 02H 功能调用一样
AH=09H	入口参数：DS：DX=欲显示的字符串逻辑地址（段地址：偏移地址） 功能说明：将指定的字符串在当前光标位置开始显示。字符串可以是任何长度，还可以包含控制字符（例如回车 0DH，换行 0AH），但必须以“\$”（其 ASCII 码为 24H）字符结尾
AH=0AH	入口参数：DS：DX=键盘输入缓冲区逻辑地址（段地址：偏移地址） 功能说明：读取从键盘输入的一个字符串（有回显），直到按下回车键。键盘输入缓冲区的第一个字节是缓冲区字节大小（最大为 255），第 2 个字节在调用结束时被功能调用填进输入的字符个数，第 3 个字节开始存放输入字符的 ASCII 码，最后是回车字符（0DH）
AH=4CH	入口参数：AL=DOS 返回码 功能说明：结束程序执行，返回 DOS

附录 C 输入输出子程序库

为了便于在汇编语言程序中进行键盘输入和显示器输出编程，本书作者编写了基本的输入输出子程序库，如表 C-1 所示。开发环境有两个文件，IO.LIB 是 16 位 DOS 环境的输入输出子程序库文件，IO.INC 是与之配合的包含文件，程序只使用了 16 位 8086 处理器指令编写。

使用输入输出子程序库的子程序，需要在源程序开头使用包含语句“INCLUDE IO.INC”声明，并且将库文件和包含文件保存在当前目录下。

这些子程序的调用方法如下：

```
MOV AX, 入口参数
CALL 子程序名
```

子程序名以 READ 开头表示键盘输入，DISP 开头表示显示器输出，参见表 C-1。中间字母 B、H、UI 和 SI 依次表示二进制、十六进制、无符号十进制和有符号十进制数；结尾字母 B 和 W 分别表示 8 位字节量和 16 位字量。另外，C 表示字符、MSG 表示字符串、R 表示寄存器。

数据输入时，二进制、十六进制和字符输入规定的位数自动结束，十进制和字符串需要用回车表示结束（超出范围显示出错 ERROR 信息，要求重新输入）。输出数据在当前光标位置开始显示，不返回任何错误信息。入口参数和出口参数都是计算机中运用的二进制数编码，有符号数用补码表示。

另外，子程序将输入参数的寄存器进行了保护，但输出参数的寄存器无法保护。如果仅返回低 8 位或低 16 位参数，高位部分不保证不会改变。输出的字符串要以 0 结尾，返回的字符串自动加入 0 作为结尾字符。

表 C-1 输入输出子程序

子程序名	参数及功能说明
READMSG	入口参数：AX=缓冲区地址。 功能说明：输入一个字符串（回车结束）。 出口参数：AX=实际输入的字符个数（不含结尾字符 0），字符串以 0 结尾
READC	出口参数：AL=字符的 ASCII 码。 功能说明：输入一个字符（回显）
DISPMSG	入口参数：AX=字符串地址。 功能说明：显示字符串（以 0 结尾）
DISPC	入口参数：AL=字符的 ASCII 码。 功能说明：显示一个字符
DISPCRLF	功能说明：光标回车换行，到下一行首个位置
READBB	出口参数：AL=8 位数据。 功能说明：输入 8 位二进制数据
READBW	出口参数：AX=16 位数据。 功能说明：输入 16 位二进制数据
DISPBB	入口参数：AL=8 位数据。 功能说明：以二进制形式显示 8 位数据
DISPBW	入口参数：AX=16 位数据。 功能说明：以二进制形式显示 16 位数据
READHB	出口参数：AL=8 位数据。 功能说明：输入 2 位十六进制数据
READHW	出口参数：AX=16 位数据。 功能说明：输入 4 位十六进制数据
DISPHB	入口参数：AL=8 位数据。 功能说明：以十六进制形式显示 2 位数据
DISPHW	入口参数：AX=16 位数据。 功能说明：以十六进制形式显示 4 位数据
READUIB	出口参数：AL=8 位数据。 功能说明：输入无符号十进制整数（≤255）

续表

子程序名	参数及功能说明
READUIW	出口参数: AX=16 位数据。 功能说明: 输入无符号十进制整数 (≤ 65535)
DISPUIB	入口参数: AL=8 位数据。 功能说明: 显示无符号十进制整数
DISPUIW	入口参数: AX=16 位数据。 功能说明: 显示无符号十进制整数
READSIB	出口参数: AL=8 位数据。 功能说明: 输入有符号十进制整数 ($-128 \sim 127$)
READSIW	出口参数: AX=16 位数据。 功能说明: 输入有符号十进制整数 ($-32768 \sim 32767$)
DISPSIB	入口参数: AL=8 位数据。 功能说明: 显示有符号十进制整数
DISPSIW	入口参数: AX=16 位数据。 功能说明: 显示有符号十进制整数
DISPRB	功能说明: 显示 8 个 8 位通用寄存器内容 (十六进制)
DISPRW	功能说明: 显示 8 个 16 位通用寄存器内容 (十六进制)
DISPRF	功能说明: 显示 6 个状态标志的状态

附录 D 列表文件符号说明

汇编过程中可以生成列表文件，其中包含伪指令生成的数据和硬指令生成的机器代码，有时需要使用一些符号表达，常用的符号含义如表 D-1 所示。

表 D-1 列表文件常见符号

符 号	含 义	示 例
=	表示符号常量等价的数值或者字符串	=000A
[]	括号之前的数值表示重复个数，括号内是重复内容	000A [24]
R	现在的地址只是相对地址（Relative）	BA 0032 R
E	现在的地址是子程序等在外模块（External）中的地址	E8 0000 E
----	汇编时无法确定的地址	BA ---- R
	操作数长度前缀指令代码（注 1）	66 8B 0E 0022 R
&	寻址方式长度前缀指令代码（注 2）	67& 8A 03
:	段超越前缀指令代码	26: A1 2000
/	字符串前缀指令代码	F3/ AB
C	源程序文件包含的汇编语句	C .model flat,stdcall
*	汇编程序生成的指令	* call ExitProcess
!	宏定义包含的指令	! xor ebx,ebx

注 1：操作数长度前缀指令代码是 66H，表示改变默认的操作数长度。

例如，32 位 Windows 操作系统默认是 32 位操作数环境，指令 MOV CX,WVAR 是 16 位操作数，所以 MASM 自动加入操作数长度前缀指令。

同样，MS-DOS 平台默认是 16 位操作数环境，指令 MOV ECX,DVAR 是 32 位操作数，所以 MASM 自动加入操作数长度前缀指令。

注 2：寻址方式长度前缀指令代码是 67H，表示改变默认的寻址方式长度。

例如，32 位 Windows 操作系统默认采用 32 位有效地址寻址方式，指令 MOV EDI, [SI]中的[SI]是 16 位有效地址寻址方式，所以 MASM 自动加入寻址方式长度前缀指令。

同样，MS-DOS 平台默认采用 16 位有效地址寻址方式，指令 MOV DI, [ESI]中的[ESI]是 32 位有效地址寻址方式，所以 MASM 自动加入寻址方式长度前缀指令。

附录 E 常见汇编错误信息

使用 ML.EXE 进行汇编过程中如果出现非法情况，会提示非法编号，并显示 ML.ERR 文件中的非法信息。

非法编号以字母 A 起头、后跟 4 位数字，形式是：Axyyy。其中 x 是非法的情况，yyy 是从 0 开始的顺序编号。

A1yyy 是致命错误（Fatal Errors），常见的致命错误信息如表 E-1 所示。

A2yyy 是严重错误（Severe Errors），常见的严重错误信息如表 E-2 所示。

A4yyy、A5yyy 和 A6yyy 依次是级别 1、2 和 3 的警告（Warnings），常见的警告信息如表 E-3 所示。

表 E-1 常见致命错误信息及中文含义

英文原文	中文含义
cannot open file	不能打开指定文件名的（源程序、包含或输出）文件
invalid command-line option	无效命令行选项（ML 无法识别给定的参数）
nesting level too deep	汇编程序达到了嵌套（20 层）的限制
line too long	源程序文件中语句行超出字符个数（512）的限制
unmatched macro nesting	模块没有结束标识符，或没有起始标识符
too many arguments	汇编程序的参数太多了
statement too complex	语句太复杂（汇编程序不能解析）
missing source filename	ML 没有找到源程序文件

表 E-2 常见严重错误信息及中文含义

英文原文	中文含义
memory operand not allowed in context	不允许存储器操作数
immediate operand not allowed	不允许立即数
extra characters after statement	语句中出现多余字符
symbol type conflict	符号类型冲突
symbol redefinition	符号重新定义
undefined symbol	无定义的符号
syntax error	语法错误
syntax error in expression	表达式中出现语法错误
invalid type expression	无效的类型表达式
.MODEL must precede this directive	该语句前必须有.MODEL 语句
expression expected	当前位置需要一个表达式
operator expected	当前位置需要一个操作符
invalid use of external symbol	外部符号的无效使用
instruction operands must be the same size	指令操作数的类型必须一致（长度相等）
instruction operand must have size	指令操作数必须有数据类型
invalid operand size for instruction	无效的指令操作数类型
constant expected	当前位置需要一个常量
operand must be a memory expression	操作数必须是一个存储器表达式
multiple base registers not allowed	不允许多个基址寄存器（例如[BX+BP]）

续表

英文原文	中文含义
multiple index registers not allowed	不允许多个变址寄存器（例如[SI+DI]）
must be index or base register	必须是基址或变址寄存器（不能是[AX]或[DX]）
invalid use of register	不能使用寄存器
DUP too complex	使用的 DUP 操作符太复杂了
invalid character in file	文件中出现无效字符
instruction prefix not allowed	不允许使用指令前缀
no operands allowed for this instruction	该指令不允许有操作数
invalid instruction operands	指令操作数无效
jump destination too far	控制转移指令的目标地址太远
cannot mix 16- and 32-bit registers	地址表达式不能既有 16 位寄存器又有 32 位寄存器
constant value too large	常量值太大了
instruction or register not accepted in current CPU mode	当前 CPU 模式不支持的指令或寄存器
END directive required at end of file	文件最后需要 END 伪指令
invalid operand for OFFSET	OFFSET 的参数无效
language type must be specified	必须指明语言类型
ORG needs a constant or local offset	ORG 语句需要一个常量或者一个局部偏移
too many operands to instruction	指令的操作数太多
macro label not defined	发现未定义的宏标号
invalid symbol type in expression	表达式中的符号类型无效
byte register cannot be first operand	字节寄存器不能作为第一个操作数
cannot use 16-bit register with a 32-bit address	不能在 32 位地址中使用 16 位寄存器
missing right parenthesis	缺少右括号
divide by zero in expression	表达式出现除以 0 的情况
INVOKE requires prototype for procedure	INVOKE 语句前需要对过程声明
missing operator in expression	表达式中缺少操作符
missing right parenthesis in expression	表达式中缺少右括号
missing left parenthesis in expression	表达式中缺少左括号
reference to forward macro definition	不能引用还没有定义的宏（先定义、后引用）
16 bit segments not allowed with /coff option	/coff 选项下不允许使用 16 位段
invalid .model parameter for flat model	无效的平展（flat）模型参数

表 E-3 常见警告信息及中文含义

英文原文	中文含义
start address on END directive ignored with .STARTUP	.STARTUP 和 END 均指明程序起始位置，END 指明的起始点被忽略
too many arguments in macro call	宏调用时的参数多于宏定义的参数
invalid command-line option value, default is used	无效命令行选项值，使用默认值
expected '>' on text literal	宏调用时参数缺少“>”符号
multiple .MODEL directives found : .MODEL ignored	发现多个.MODEL 语句，只使用第一个.MODEL 语句
@@: label defined but not referenced	定义了标号，但没有被访问
types are different	INVOKE 语句的类型不同于声明语句，汇编程序进行适当转换
calling convention not supported in flat model	平展（flat）模型下不支持的调用规范
no return from procedure	PROC 生成起始代码，但在其过程中没有 RET 或 IRET 指令

附录 F 通用指令列表

表 F-1 指令符号说明

符 号	说 明
r8	任意一个 8 位通用寄存器 AH/AL/BH/BL/CH/CL/DH/DL
r16	任意一个 16 位通用寄存器 AX/BX/CX/DX/SI/DI/BP/SP
r32	任意一个 32 位通用寄存器 EAX/EBX/ECX/EDX/ESI/EDI/EBP/ESP
reg	代表 r8/r16/r32
seg	段寄存器 CS/DS/ES/SS 和 FS/GS
m8	一个 8 位存储器操作数单元
m16	一个 16 位存储器操作数单元
m32	一个 32 位存储器操作数单元
mem	代表 m8/m16/m32
i8	一个 8 位立即数
i16	一个 16 位立即数
i32	一个 32 位立即数
imm	代表 i8/i16/i32
dest	目的操作数
src	源操作数
label	标号

表 F-2 16/32 位基本指令的汇编格式

指令类型	指令汇编格式	指令功能简介
传送指令	MOV reg/mem,imm	dest←src
	MOV reg/mem/seg,reg	
	MOV reg/seg,mem	
	MOV reg/mem,seg	
交换指令	XCHG reg,reg/mem	reg↔reg/mem
	XCHG reg/mem,reg	
转换指令	XLAT buffer	AL←DS:[(E)BX+AL]
	XLAT	注：buffer 表示代码表所在的缓冲区
堆栈指令	PUSH reg/mem/seg	寄存器/存储器入栈
	PUSH imm	立即数入栈
	POP reg/seg/mem	出栈
	PUSHA	保护所有 r16
	POPA	恢复所有 r16
	PUSHAD	保护所有 r32
	POPAD	恢复所有 r32
标志传送	LAHF	AH←FLAG 低字节
	SAHF	FLAG 低字节←AH
	PUSHF	FLAGS 入栈
	POPF	FLAGS 出栈
	PUSHFD	EFLAGS 入栈
	POPFD	EFLAGS 出栈

续表

指令类型	指令汇编格式	指令功能简介
地址传送	LEA r16/r32,mem LDS r16/r32,mem LES r16/r32,mem LFS r16/r32,mem LGS r16/r32,mem LSS r16/r32,mem	r16/r32 ← 16/32 位有效地址 DS: r16/r32 ← 32/48 位远指针 ES: r16/r32 ← 32/48 位远指针 FS: r16/r32 ← 32/48 位远指针 GS: r16/r32 ← 32/48 位远指针 SS: r16/r32 ← 32/48 位远指针
输入输出	IN AL/AX/EAX,i8/DX OUT i8/DX,AL/AX/EAX	AL/AX/EAX ← I/O 端口 i8/[DX] I/O 端口 i8/[DX] ← AL/AX/EAX
加法运算	ADD reg,imm/reg/mem ADD mem,imm/reg ADC reg,imm/reg/mem ADC mem,imm/reg INC reg/mem	ADD: dest ← dest + src ADC: dest ← dest + src + CF INC: reg/mem ← reg/mem + 1
减法运算	SUB reg,imm/reg/mem SUB mem,imm/reg SBB reg,imm/reg/mem SBB mem,imm/reg DEC reg/mem NEG reg/mem CMP reg,imm/reg/mem CMP mem,imm/reg	SUB: dest ← dest - src SBB: dest ← dest - src - CF DEC: reg/mem ← reg/mem - 1 NEG: reg/mem ← 0 - reg/mem CMP: dest - src
乘法运算	MUL reg/mem IMUL reg/mem IMUL r16,r16/m16/i8/i16 IMUL r16,r/m16,i8/i16 IMUL r32,r32/m32/i8/i32 IMUL r32,r32/m32,i8/i32	无符号数值乘法 有符号数值乘法 r16 ← r16 × r16/m16/i8/i16 r16 ← r/m16 × i8/i16 r32 ← r32 × r32/m32/i8/i32 r32 ← r32/m32 × i8/i32
除法运算	DIV reg/mem IDIV reg/mem	无符号数值除法 有符号数值除法
符号扩展	CBW CWD CWDE CDQ MOVSX r16,r8/m8 MOVSX r32,r8/m8/r16/m16 MOVZX r16,r8/m8 MOVZX r32,r8/m8/r16/m16	把 AL 符号扩展为 AX 把 AX 符号扩展为 DX.AX 把 AX 符号扩展为 EAX 把 EAX 符号扩展为 EDX.EAX 把 r8/m8 符号扩展并传送至 r16 把 r8/m8/r16/m16 符号扩展并传送至 r32 把 r8/m8 零位扩展并传送至 r16 把 r8/m8/r16/m16 零位扩展并传送至 r32
十进制调整	DAA DAS AAA AAS AAM AAD	将 AL 中的加和调整为压缩 BCD 码 将 AL 中的减差调整为压缩 BCD 码 将 AL 中的加和调整为非压缩 BCD 码 将 AL 中的减差调整为非压缩 BCD 码 将 AX 中的乘积调整为非压缩 BCD 码 将 AX 中的非压缩 BCD 码扩展成二进制数

续表

指令类型	指令汇编格式	指令功能简介
逻辑运算	AND reg,imm/reg/mem AND mem,imm/reg OR reg,imm/reg/mem OR mem,imm/reg XOR reg,imm/reg/mem XOR mem,imm/reg TEST reg,imm/reg/mem TEST mem,imm/reg NOT reg/mem	AND: $\text{dest} \leftarrow \text{dest AND src}$ OR: $\text{dest} \leftarrow \text{dest OR src}$ XOR: $\text{dest} \leftarrow \text{dest XOR src}$ TEST: dest AND src NOT: $\text{reg/mem} \leftarrow \text{NOT reg/mem}$
移位	SAL reg/mem,1/CL/i8 SAR reg/mem,1/CL/i8 SHL reg/mem,1/CL/i8 SHR reg/mem,1/CL/i8	算术左移 1/CL/i8 指定的位数 算术右移 1/CL/i8 指定的位数 与 SAL 相同 逻辑右移 1/CL/i8 指定的位数
循环移位	ROL reg/mem,1/CL/i8 ROR reg/mem,1/CL/i8 RCL reg/mem,1/CL/i8 RCR reg/mem,1/CL/i8	循环左移 1/CL/i8 指定的位数 循环右移 1/CL/i8 指定的位数 带进位循环左移 1/CL/i8 指定的位数 带进位循环右移 1/CL/i8 指定的位数
串操作	MOVS[B/W/D] LODS[B/W/D] STOS[B/W/D] CMPS[B/W/D] SCAS[B/W/D] INS[B/W/D] OUTS[B/W/D] REP REPZ / REPE REPNZ / REPNE	串传送 串读取 串存储 串比较 串扫描 I/O 串输入 I/O 串输出 重复前缀 相等重复前缀 不等重复前缀
转移	JMP label JMP r16/r32/m16 JCC label JCXZ label JECXZ label	无条件直接转移 无条件间接转移 条件转移 CX 等于 0 转移 ECX 等于 0 转移
循环	LOOP label LOOPZ / LOOPE label LOOPNZ / LOOPNE label	(E)CX \leftarrow (E)CX - 1; 若 (E)CX \neq 0, 循环 (E)CX \leftarrow (E)CX - 1; 若 (E)CX \neq 0 且 ZF=1, 循环 (E)CX \leftarrow (E)CX - 1; 若 (E)CX \neq 0 且 ZF=0, 循环
子程序	CALL label CALL r16/m16 RET RET i16	直接调用 间接调用 无参数返回 有参数返回
中断	INT i8 IRET INTO	中断调用 中断返回 溢出中断调用

续表

指令类型	指令汇编格式	指令功能简介
高级语言支持	ENTER i16,i8 LEAVE BOUND r16/r32,mem	建立堆栈帧 释放堆栈帧 边界检测
处理器控制	CLC STC CMC CLD STD CLI STI NOP WAIT HLT LOCK SEG:	CF←0 CF←1 CF←~CF DF←0 DF←1 IF←0 IF←1 空操作指令 等待指令 停机指令 封锁前缀 段超越前缀
保护方式类指令	略	略

表 F-3 新增 32 位指令的汇编格式

指令类型	指令汇编格式	指令功能简介
双精度移位	SHLD r16/r32/m16/m32,r16/r32,i8/CL SHRD r16/r32/m16/m32,r16/r32,i8/CL	将 r16/r32 的 i8/CL 位左移进入 r16/r32/m16/m32 将 r16/r32 的 i8/CL 位右移进入 r16/r32/m16/m32
位扫描	BSF r16/r32,r16/r32/m16/m32 BSR r16/r32,r16/r32/m16/m32	前向扫描 后向扫描
位测试	BT r16/r32,i8/r16/r32 BTC r16/r32,i8/r16/r32 BTR r16/r32,i8/r16/r32 BTS r16/r32,i8/r16/r32	测试位 测试位求反 测试位复位 测试位置位
条件设置	SETcc r8/m8	条件成立, r8/m8=1; 否则, r8/m8=0
系统寄存器传送	MOV CRn/DRn/TRn,r32 MOV r32,CRn/DRn/TRn	装入系统寄存器 读取系统寄存器
多处理器	BSWAP r32 XADD reg/mem,reg CMPXCHG reg/mem,reg	字节交换 交换加 比较交换
高速缓存	INVD WBINVD INVLPG mem	高速缓存无效 回写及高速缓存无效 TLB 无效
Pentium 指令	CMPXCHG8B m64 CPUID RDTSC RDMSR WRMSR RSM	8 字节比较交换 (m64 表示 64 位存储器操作数) 返回处理器的有关特征信息 EDX.EAX←64 位时间标记计数器值 EDX.EAX←模型专用寄存器值 模型专用寄存器值←EDX.EAX 从系统管理方式返回
Pentium Pro 指令	CMOVcc r16/r32,r16/r32/m16/m32 RDPMSR UD2	条件成立, r16/r32←r16/r32/m16/m32 EDX.EAX←40 位性能监控计数器值 产生一个无效操作码异常

附录 G MASM 伪指令和操作符列表

表 G-1 MASM 6.11 的主要伪指令

伪指令类型	伪 指 令
变量定义	DB/BYTE/SBYTE、DW/WORD/SWORD、DD/DWORD/SDWORD/REAL4 FWORD/DF、QWORD/DQ/REAL8、TBYTE/DT/REAL10
定位	EVEN、ALIGN、ORG
符号定义	RADIX、=、EQU、TEXTEQU、LABEL
简化段定义	.MODEL、.STARTUP、.EXIT、.CODE、.STACK、.DATA、.DATA?、.CONST .FARDATA、.FARDATA?
完整段定义	SEGMENT/ENDS、GROUP、ASSUME、END、.DOSSEG/.ALPHA/.SEQ
复杂数据类型	STRUCT/STRUC、UNION、RECORD、TYPEDEF、ENDS
流程控制	.IF/.ELSE/.ELSEIF/.ENDIF、.WHILE/.ENDW、.REPEAT/.UNTIL[CXZ]、.BREAK/.CONTINUE
过程定义	PROC/ENDP、PROTO、INVOKE
宏汇编	MACRO/ENDM、PURGE、LOCAL、PUSHCONTEXT、POPCONTEXT、EXITM、GOTO
重复汇编	REPEAT/REPT、WHILE、FOR/IRP、FORC/IRPC
条件汇编	IF/IFE、IFB/IFNB、IFDEF/IFNDEF/IFDIF/IFIDN、ELSE、ELSEIF、ENDIF
模块化	PUBLIC、EXTEN/EXTERN[DEF]、COMM、INCLUDE、INCLUDELIB
条件错误	.ERR/ERRE、.ERRB/ERRNB、.ERRDEF/ERRNDEF、.ERRDIF/ERRIDN
列表控制	TITLE/SUBTITLE、PAGE、.LIST/.LISTALL/.LISTMACRO/.LISTMACROALL/.LISTIF .NOLIST、.TFCOND、.CREF/.NOCREF、COMMENT、ECHO
处理器选择	.8086、.186、.286/.286P、.386/.386P、.486/.486P、.8087、.287、.387、.NO87
字符串处理	CATSTR、INSTR、SIZESTR、SUBSTR

表 G-2 MASM 6.11 的主要操作符

操作符类型	操 作 符
算术运算符	+, -, *, /, MOD
逻辑运算符	AND, OR, XOR, NOT
移位运算符	SHL, SHR
关系运算符	EQ, NE, GT, LT, GE, LE
高低分离符	HIGH, LOW, HIGHWORD, LOWWORD
地址操作符	[], \$, :, OFFSET, SEG
类型操作符	PTR, THIS, SHORT, TYPE, SIZEOF/SIZE, LENGTHOF/LENGTH
复杂数据操作符	(), <>, ., MASK, WIDTH, ?, DUP, ', "
宏操作符	&, <>, !, %, ;;
流程条件操作符	==, !=, >, >=, <, <=, &&, , !, & CARRY?, OVERFLOW?, PARITY?, SIGN?, ZERO?
预定义符号	@CatStr, @code, @CodeSize, @Cpu, @CurSeg, @data, @DataSize, @Date @Environ, @fardata, @fardata?, @FileCur, @FileName, @InStr, @Interface @Line, @Model, @SizeStr, @SubStr, @stack, @Time, @Version, @WordSize

参 考 文 献

- [1] 钱晓捷. 汇编语言程序设计 (第 4 版). 电子工业出版社, 2012.
- [2] 钱晓捷. 微型计算机原理及应用 (第 2 版). 清华大学出版社, 2011.
- [3] 钱晓捷. 微型计算机原理及应用教学辅导与习题解答 (第 2 版). 清华大学出版社, 2011.
- [4] 钱晓捷. 32 位汇编语言程序设计. 机械工业出版社, 2011.
- [5] Kip R. Irvine. 温玉杰等译. Intel 汇编语言程序设计 (第五版). 电子工业出版社, 2007.
- [6] Intel, Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture (253665.pdf), <http://developer.intel.com>, 2010.
- [7] Richard C. Detmer. 80x86 汇编语言与计算机体系结构 (英文版). 机械工业出版社, 2004.
- [8] Barry B.Brey. Intel 微处理器 (英文影印版 • 第 7 版). 机械工业出版社, 2006.



汇编语言简明教程

目 录	内容提要 (教学重点)
第1章 汇编语言基础	在了解个人计算机软硬件系统的基础上,熟悉通用寄存器和存储器组织,掌握汇编语言的语句格式、程序框架和开发方法
第2章 数据表示和寻址	在理解计算机如何表达数据的基础上,熟悉汇编语言中如何使用常量和变量,掌握处理器指令寻址数据的方式
第3章 通用数据处理指令	熟悉处理器数据传送、算术运算、逻辑运算和移位操作等基本指令,通过程序片段掌握指令功能和编程应用
第4章 程序结构	以顺序、分支和循环程序结构为主线,结合数值运算、数组处理等示例程序,掌握控制转移指令以及编写基本程序结构的方法
第5章 模块化程序设计	以子程序结构为主体,围绕二、十六和十进制数码转换实现键盘输入和显示输出,熟悉子程序、文件包含、宏汇编等各种多模块编程的思想
第6章 32位指令及其编程	了解80x86系列处理器发展概况,理解32位处理器的运行环境,熟悉新增的32位特色指令,掌握DOS环境使用32位指令的特点
第7章 Windows编程	熟悉汇编语言调用API函数的方法,掌握控制台输入输出函数。了解MASM的高级特性,以及Windows图形窗口程序的编写
第8章 与Visual C++的混合编程	了解嵌入汇编和模块连接进行混合编程方法,理解堆栈帧的作用,熟悉汇编语言调用高级语言函数和开发调试过程
第9章 浮点、多媒体及64位指令	理解浮点数据格式、多媒体数据格式及64位编程环境的特点,了解浮点、多媒体和64位等特色指令

- 删繁就简、重点明确的教学内容:选择处理器通用的基本指令和反映汇编语言特色的常用伪指令;侧重指令功能和编程思想、特别介绍相关硬件工作原理。
- 贯穿始终、突出实践的教学过程:将上机实践贯穿始终,教学内容融入了约80个例题程序和约70个习题程序。循序渐进、深入浅出的教学原则:以“循序渐进、难点分散、前后对照”为原则,做到“语言浅显、描述详尽、图表准确”。
- 教学网站:“大学微机技术系列课程教学辅助网站”(http://www2.zzu.edu.cn/qwfw),提供本书的教学课件(电子教案)、配套软件(含示例源程序文件)等辅助资源。



策划编辑:章海涛
责任编辑:章海涛
封面设计:徐海燕

ISBN 978-7-121-20184-4



9 787121 201844 >

定价:39.00元